BOOK 18: THREE.JS COMPLETE
Chapter 1: What is Three.js

THREE.JS EXPLAINED

Three.js is a JavaScript library for 3D graphics in the browser.

It simplifies WebGL which is the low-level API for GPU-accelerated graphics.

Without Three.js you write hundreds of lines of shader code and matrix math.

With Three.js you write a few lines of JavaScript.

WHY THREE.JS EXISTS

WebGL is powerful but extremely complex.

Drawing a single triangle in raw WebGL requires:
- Vertex shader code
- Fragment shader code
- Buffer creation and binding
- Attribute and uniform setup
- Matrix transformations
- State management

Three.js abstracts all of this into simple objects.

Instead of shaders you use materials.
Instead of buffers you use geometries.
Instead of matrices you use position rotation scale.

WHAT YOU CAN BUILD

Product configurators with 3D models.

Data visualizations in three dimensions.

Games with physics and interactions.

Architectural walkthroughs.

Virtual reality experiences.

Artistic and creative visualizations.

Scientific simulations.

Portfolio websites with 3D elements.

THE THREE CORE CONCEPTS

Every Three.js application needs three things.

Scene - The container for all 3D objects.

Think of it as a stage where everything happens.

```
let scene = new THREE.Scene()
```

Camera - The viewpoint into the scene.

Determines what you see and from where.

```
let camera = new THREE.PerspectiveCamera(75, width / height, 0.1, 1000)
```

Renderer - Draws the scene from the camera view.

Creates the actual pixels on screen.

```
let renderer = new THREE.WebGLRenderer()
```

MINIMAL EXAMPLE

This creates a spinning cube.

```
let scene = new THREE.Scene()

let camera = new THREE.PerspectiveCamera(
 75,
 window.innerWidth / window.innerHeight,
 0.1,
 1000
)

let renderer = new THREE.WebGLRenderer()
renderer.setSize(window.innerWidth, window.innerHeight)
document.body.appendChild(renderer.domElement)

let geometry = new THREE.BoxGeometry(1, 1, 1)
let material = new THREE.MeshBasicMaterial({ color: 0x00ff00 })
let cube = new THREE.Mesh(geometry, material)
scene.add(cube)

camera.position.z = 5

function animate() {
 requestAnimationFrame(animate)
 cube.rotation.x += 0.01
 cube.rotation.y += 0.01
 renderer.render(scene, camera)
```

```
}

animate()
```

Breaking down what happens:

Scene created as empty container.
Camera created with 75 degree field of view.
Renderer created and canvas added to page.
Box geometry defines the shape.
Basic material defines the color.
Mesh combines geometry and material.
Mesh added to scene.
Camera moved back to see the cube.
Animation loop rotates cube and renders.

SETTING UP A PROJECT

Method 1 - CDN for quick experiments:

Create an HTML file.

index.html:

```
<!DOCTYPE html>
<html>
<head>
 <title>Three.js</title>
 <style>
 body { margin: 0; }
 canvas { display: block; }
 </style>
</head>
<body>
 <script src="https://cdnjs.cloudflare.com/ajax/libs/three.js/
r128/three.min.js"></script>
```

```
 <script src="main.js"></script>
</body>
</html>
```

Method 2 - npm for real projects:

```
npm init -y
npm install three
```

Then in your JavaScript:

```
import * as THREE from "three"
```

Method 3 - Vite for modern development:

```
npm create vite@latest my-threejs-app
cd my-threejs-app
npm install three
npm run dev
```

THE COORDINATE SYSTEM

Three.js uses a right-handed coordinate system.

X axis points right (red).
Y axis points up (green).
Z axis points toward you (blue).

Position (0, 0, 0) is the center of the scene.

Positive X moves right.
Negative X moves left.
Positive Y moves up.
Negative Y moves down.
Positive Z moves toward camera.

Negative Z moves away from camera.

UNITS AND SCALE

Three.js has no built-in units.

1 unit can be 1 meter or 1 centimeter or 1 mile.

You decide what units mean for your project.

For most projects treat 1 unit as 1 meter.

This works well with:
- Physics engines
- VR/AR applications
- Real-world scale models

BROWSER SUPPORT

Three.js requires WebGL support.

All modern browsers support WebGL:
- Chrome
- Firefox
- Safari
- Edge

Mobile browsers also support WebGL.

Very old browsers may not work.

You can check for WebGL support:

```
if (WebGLRenderingContext) {
```

```
 console.log("WebGL supported")
} else {
 console.log("WebGL not supported")
}
```

PERFORMANCE CONSIDERATIONS

3D graphics are GPU intensive.

More objects means more work.
Higher resolution means more pixels.
Complex materials need more calculations.

Start simple and add complexity.

Monitor frame rate during development.

Target 60 frames per second for smooth animation.

THE RENDER LOOP

Three.js does not automatically update the screen.

You must call render manually.

For static scenes render once:

```
renderer.render(scene, camera)
```

For animated scenes use a loop:

```
function animate() {
 requestAnimationFrame(animate)
```

```
 renderer.render(scene, camera)
}

animate()
```

requestAnimationFrame syncs with monitor refresh rate.

This gives smooth 60fps animation.

HANDLING WINDOW RESIZE

When browser window resizes the canvas should resize too.

```
window.addEventListener("resize", function() {
 camera.aspect = window.innerWidth / window.innerHeight
 camera.updateProjectionMatrix()
 renderer.setSize(window.innerWidth, window.innerHeight)
})
```

camera.aspect updates the aspect ratio.
updateProjectionMatrix recalculates the camera.
setSize resizes the canvas.

PRACTICE

Create a new project with Three.js.

Set up scene camera and renderer.

Add a simple cube to the scene.

Make the cube rotate.

Handle window resize.

WHAT YOU LEARNED

You learned what Three.js is and why it exists.

You learned the three core concepts: scene camera renderer.

You learned how to set up a project.

You learned the coordinate system.

You learned about the render loop.

The next chapter teaches scenes in detail.
BOOK 18: THREE.JS COMPLETE
Chapter 2: The Scene

WHAT IS A SCENE

The scene is the container for everything visible.

All objects lights and cameras live in the scene.

Think of it as a 3D world or stage.

let scene = new THREE.Scene()

SCENE PROPERTIES

Background color:

```
scene.background = new THREE.Color(0x000000)
```

This sets a solid black background.

Any hex color works:

```
scene.background = new THREE.Color(0x87ceeb)
```

Sky blue background.

Background with texture:

```
let loader = new THREE.TextureLoader()
let texture = loader.load("background.jpg")
scene.background = texture
```

Background with cube map for 360 environments:

```
let cubeLoader = new THREE.CubeTextureLoader()
let cubeTexture = cubeLoader.load([
 "px.jpg", "nx.jpg",
 "py.jpg", "ny.jpg",
 "pz.jpg", "nz.jpg"
])
scene.background = cubeTexture
```

FOG

Fog makes distant objects fade into a color.

Creates depth and atmosphere.

Linear fog fades linearly between near and far:

```
scene.fog = new THREE.Fog(0xcccccc, 10, 100)
```

First parameter is fog color.
Second is where fog starts (10 units from camera).
Third is where fog is complete (100 units).

Exponential fog for more realistic effect:

```
scene.fog = new THREE.FogExp2(0xcccccc, 0.02)
```

First parameter is color.
Second is density (higher means thicker fog).

ADDING OBJECTS TO SCENE

Use the add method:

```
let cube = new THREE.Mesh(geometry, material)
scene.add(cube)
```

Add multiple objects:

```
scene.add(cube)
scene.add(sphere)
scene.add(light)
```

Add multiple at once:

```
scene.add(cube, sphere, light)
```

REMOVING OBJECTS FROM SCENE

Use the remove method:

scene.remove(cube)

Remove multiple:

scene.remove(cube, sphere)

THE SCENE GRAPH

Objects can have children.

Children move with their parent.

```
let group = new THREE.Group()
group.add(cube1)
group.add(cube2)
scene.add(group)
```

Now rotating the group rotates both cubes together.

Real example - a car:

```
let car = new THREE.Group()

let body = new THREE.Mesh(bodyGeometry, bodyMaterial)
car.add(body)

let wheel1 = new THREE.Mesh(wheelGeometry, wheelMaterial)
wheel1.position.set(-1, -0.5, 1)
car.add(wheel1)
```

```
let wheel2 = new THREE.Mesh(wheelGeometry, wheelMaterial)
wheel2.position.set(1, -0.5, 1)
car.add(wheel2)

let wheel3 = new THREE.Mesh(wheelGeometry, wheelMaterial)
wheel3.position.set(-1, -0.5, -1)
car.add(wheel3)

let wheel4 = new THREE.Mesh(wheelGeometry, wheelMaterial)
wheel4.position.set(1, -0.5, -1)
car.add(wheel4)

scene.add(car)
```

Moving car.position moves the entire car with all wheels.

TRAVERSING THE SCENE

Loop through all objects:

```
scene.traverse(function(object) {
 console.log(object.name)
})
```

Find objects by name:

```
let cube = scene.getObjectByName("myCube")
```

Find by property:

```
scene.traverse(function(object) {
 if (object.isMesh) {
 object.material.color.set(0xff0000)
 }
```

})

This turns all meshes red.

OBJECT PROPERTIES

Every object in the scene has these properties:

Position - where the object is:

```
cube.position.x = 5
cube.position.y = 2
cube.position.z = -3
```

Or set all at once:

```
cube.position.set(5, 2, -3)
```

Rotation - how the object is rotated:

```
cube.rotation.x = Math.PI / 4
cube.rotation.y = Math.PI / 2
cube.rotation.z = 0
```

Rotation uses radians not degrees.

To convert degrees to radians:

```
let radians = degrees * (Math.PI / 180)
```

Scale - how big the object is:

```
cube.scale.x = 2
cube.scale.y = 0.5
```

cube.scale.z = 1

Or set all at once:

cube.scale.set(2, 0.5, 1)

This makes the cube twice as wide and half as tall.

Visibility:

cube.visible = false

Hides the object but keeps it in scene.

cube.visible = true

Shows it again.

NAMING OBJECTS

Give objects names for easy reference:

cube.name = "playerCube"

let player = scene.getObjectByName("playerCube")

LAYERS

Objects can be on different layers.

Cameras only see objects on their layers.

Useful for hiding objects from certain cameras.

```
cube.layers.set(1)
camera.layers.enable(1)
```

Default layer is 0.

CLEARING THE SCENE

Remove all objects:

```
while(scene.children.length > 0) {
 scene.remove(scene.children[0])
}
```

Or dispose properly to free memory:

```
function clearScene() {
 scene.traverse(function(object) {
 if (object.geometry) {
 object.geometry.dispose()
 }
 if (object.material) {
 if (Array.isArray(object.material)) {
 object.material.forEach(function(material) {
 material.dispose()
 })
 } else {
 object.material.dispose()
 }
 }
 })

 while(scene.children.length > 0) {
 scene.remove(scene.children[0])
 }
}
```

SCENE HELPERS

Axes helper shows X Y Z directions:

```
let axesHelper = new THREE.AxesHelper(5)
scene.add(axesHelper)
```

Red is X. Green is Y. Blue is Z.
Parameter is the size of the axes.

Grid helper shows a ground grid:

```
let gridHelper = new THREE.GridHelper(10, 10)
scene.add(gridHelper)
```

First parameter is size.
Second is number of divisions.

PRACTICE

Create a scene with a background color.

Add fog to the scene.

Create a group of objects.

Use traverse to modify objects.

Add helpers for debugging.

WHAT YOU LEARNED

You learned how to create and configure scenes.

You learned about backgrounds and fog.

You learned the scene graph and groups.

You learned object properties: position rotation scale.

You learned to traverse and find objects.

The next chapter teaches cameras.

BOOK 18: THREE.JS COMPLETE

Chapter 3: Cameras

WHAT IS A CAMERA

The camera defines what you see.

It has a position in the scene and looks in a direction.

What the camera sees gets rendered to the screen.

PERSPECTIVE CAMERA

Most common camera type.

Objects farther away appear smaller.

This matches how human eyes work.

let camera = new THREE.PerspectiveCamera(fov, aspect, near, far)

fov - Field of view in degrees.
How wide the camera sees.
Human eye is about 75 degrees.
Lower values zoom in.
Higher values show more but distort edges.

aspect - Width divided by height.
Usually window.innerWidth / window.innerHeight.
Must match canvas proportions.

near - Closest distance rendered.
Objects closer than this are not visible.
Usually 0.1.

far - Farthest distance rendered.
Objects beyond this are not visible.
Usually 1000 or more.

Example:

```
let camera = new THREE.PerspectiveCamera(
 75,
 window.innerWidth / window.innerHeight,
 0.1,
 1000
)
```

ORTHOGRAPHIC CAMERA

No perspective distortion.

Objects stay same size regardless of distance.

Used for 2D games, CAD, and technical views.

let camera = new THREE.OrthographicCamera(left, right, top, bottom, near, far)

left - Left edge of view.
right - Right edge of view.
top - Top edge of view.
bottom - Bottom edge of view.
near - Closest distance rendered.
far - Farthest distance rendered.

Example:

```
let aspect = window.innerWidth / window.innerHeight
let viewSize = 10

let camera = new THREE.OrthographicCamera(
 -viewSize * aspect,
 viewSize * aspect,
 viewSize,
 -viewSize,
 0.1,
 1000
)
```

CAMERA POSITION

Position the camera in the scene:

```
camera.position.x = 0
camera.position.y = 5
camera.position.z = 10
```

Or use set:

camera.position.set(0, 5, 10)

This puts the camera 5 units up and 10 units back.

CAMERA DIRECTION

By default camera looks toward negative Z.

Use lookAt to point at something:

camera.lookAt(0, 0, 0)

This points the camera at the origin.

Look at an object:

camera.lookAt(cube.position)

Look at specific point:

camera.lookAt(new THREE.Vector3(5, 2, 0))

CAMERA MOVEMENT

Move camera in animation loop:

```
function animate() {
 requestAnimationFrame(animate)

 camera.position.x = Math.sin(Date.now() * 0.001) * 10
 camera.position.z = Math.cos(Date.now() * 0.001) * 10
```

```
 camera.lookAt(0, 0, 0)

 renderer.render(scene, camera)
}
```

This orbits the camera around the origin.

CAMERA AS CHILD

Attach camera to an object:

```
let player = new THREE.Mesh(geometry, material)
player.add(camera)
camera.position.set(0, 2, 5)
scene.add(player)
```

Camera follows the player.

Moving player moves the camera too.

UPDATING CAMERA ON RESIZE

When window resizes update the camera:

```
window.addEventListener("resize", function() {
 camera.aspect = window.innerWidth / window.innerHeight
 camera.updateProjectionMatrix()
 renderer.setSize(window.innerWidth, window.innerHeight)
})
```

updateProjectionMatrix must be called after changing camera properties.

MULTIPLE CAMERAS

You can have multiple cameras.

Render different views:

```
let mainCamera = new THREE.PerspectiveCamera(75, aspect, 0.1, 1000)
let topCamera = new THREE.OrthographicCamera(-10, 10, 10, -10, 0.1, 1000)

topCamera.position.set(0, 50, 0)
topCamera.lookAt(0, 0, 0)
```

Render split screen:

```
renderer.setViewport(0, 0, width / 2, height)
renderer.render(scene, mainCamera)

renderer.setViewport(width / 2, 0, width / 2, height)
renderer.render(scene, topCamera)
```

CAMERA HELPER

Visualize camera frustum for debugging:

```
let helper = new THREE.CameraHelper(camera)
scene.add(helper)
```

Shows the camera view cone.

Useful when setting up multiple cameras.

FIELD OF VIEW EFFECTS

Low FOV (30-40):
- Telephoto lens effect
- Objects appear closer together
- Less distortion

Medium FOV (60-75):
- Natural human-like view
- Good for most applications

High FOV (90-120):
- Wide angle lens effect
- More visible but distorted edges
- Can cause motion sickness in VR

NEAR AND FAR PLANES

Keep near as high as possible.
Keep far as low as possible.

This improves depth precision.

Bad:
near = 0.0001, far = 100000

Good:
near = 0.1, far = 1000

Large range causes z-fighting where surfaces flicker.

ZOOM

Perspective camera:

```
camera.zoom = 2
camera.updateProjectionMatrix()
```

Higher zoom magnifies the view.

Orthographic camera:

Adjust the view size instead:

```
camera.left = -5
camera.right = 5
camera.top = 5
camera.bottom = -5
camera.updateProjectionMatrix()
```

CAMERA MATRICES

Camera has matrices for advanced use:

camera.matrixWorld - Position and rotation in world space.
camera.projectionMatrix - How 3D maps to 2D.
camera.projectionMatrixInverse - Reverse of projection.

Used for raycasting and screen-to-world calculations.

PRACTICAL CAMERA SETUP

First person view:

```
camera.position.set(0, 1.7, 0)
```

Position at eye height.

Third person view:

```
camera.position.set(0, 5, 10)
camera.lookAt(player.position)
```

Top down view:

```
camera.position.set(0, 20, 0)
camera.lookAt(0, 0, 0)
camera.rotation.z = 0
```

Isometric view:

```
let camera = new THREE.OrthographicCamera(-10, 10, 10, -10,
0.1, 1000)
camera.position.set(10, 10, 10)
camera.lookAt(0, 0, 0)
```

PRACTICE

Create a perspective camera.

Position it to see your scene.

Make it orbit around the origin.

Create an orthographic camera.

Set up window resize handling.

WHAT YOU LEARNED

You learned perspective and orthographic cameras.

You learned camera positioning and lookAt.

You learned about FOV near and far planes.

You learned to update camera on resize.

You learned multiple cameras and helpers.

The next chapter teaches renderers.
BOOK 18: THREE.JS COMPLETE
Chapter 4: Renderers

WHAT IS A RENDERER

The renderer draws the scene.

It takes the scene and camera and produces an image.

The image displays in a canvas element.

```
let renderer = new THREE.WebGLRenderer()
```

BASIC SETUP

Create renderer and add to page:

```
let renderer = new THREE.WebGLRenderer()
```

```
renderer.setSize(window.innerWidth, window.innerHeight)
document.body.appendChild(renderer.domElement)
```

renderer.domElement is the canvas element.

setSize sets the canvas width and height.

WEBGLRENDERER OPTIONS

Pass options when creating:

```
let renderer = new THREE.WebGLRenderer({
 antialias: true,
 alpha: true,
 preserveDrawingBuffer: true
})
```

antialias - Smooth edges. Default false.
Reduces jagged edges on diagonal lines.
Has performance cost.

alpha - Transparent background. Default false.
Lets page background show through.

preserveDrawingBuffer - Keep canvas content. Default false.
Required for taking screenshots.
Has performance cost.

USING EXISTING CANVAS

Instead of creating a new canvas use an existing one:

HTML:

<canvas id="myCanvas"></canvas>

JavaScript:

```
let canvas = document.getElementById("myCanvas")
let renderer = new THREE.WebGLRenderer({ canvas: canvas })
```

PIXEL RATIO

High DPI displays have more pixels per inch.

Set pixel ratio for sharp rendering:

renderer.setPixelRatio(window.devicePixelRatio)

Or limit for performance:

renderer.setPixelRatio(Math.min(window.devicePixelRatio, 2))

This caps at 2x even on 3x displays.

RENDER SIZE

Set the size of the output:

renderer.setSize(800, 600)

Full window:

renderer.setSize(window.innerWidth, window.innerHeight)

Get current size:

```
let size = new THREE.Vector2()
renderer.getSize(size)
console.log(size.x, size.y)
```

RENDERING

Render the scene:

```
renderer.render(scene, camera)
```

Call this every frame for animation:

```
function animate() {
 requestAnimationFrame(animate)
 renderer.render(scene, camera)
}

animate()
```

CLEAR COLOR

Set background color when no scene background:

```
renderer.setClearColor(0x000000)
```

Black background.

```
renderer.setClearColor(0x87ceeb)
```

Sky blue background.

With transparency:

renderer.setClearColor(0x000000, 0)

Fully transparent.

renderer.setClearColor(0x000000, 0.5)

50% transparent black.

AUTO CLEAR

By default renderer clears before each render.

Disable to accumulate frames:

renderer.autoClear = false

Manual clear:

renderer.clear()

Clear specific buffers:

renderer.clear(true, true, false)

Parameters are color, depth, stencil.

SHADOW MAPS

Enable shadow rendering:

renderer.shadowMap.enabled = true

Shadow map types:

renderer.shadowMap.type = THREE.BasicShadowMap

Fast but blocky shadows.

renderer.shadowMap.type = THREE.PCFShadowMap

Smoother shadows. Default.

renderer.shadowMap.type = THREE.PCFSoftShadowMap

Softer shadows. More expensive.

renderer.shadowMap.type = THREE.VSMShadowMap

Very soft shadows. Most expensive.

TONE MAPPING

Controls how colors are mapped to screen.

renderer.toneMapping = THREE.NoToneMapping

No adjustment. Default.

renderer.toneMapping = THREE.LinearToneMapping

Linear mapping.

renderer.toneMapping = THREE.ReinhardToneMapping

Filmic look.

renderer.toneMapping = THREE.CineonToneMapping

Cinema style.

renderer.toneMapping = THREE.ACESFilmicToneMapping

Modern film look. Popular choice.

Exposure adjustment:

renderer.toneMappingExposure = 1.0

Higher values brighten. Lower values darken.

OUTPUT ENCODING

How colors are encoded:

renderer.outputEncoding = THREE.sRGBEncoding

Correct for most displays. Use this.

renderer.outputEncoding = THREE.LinearEncoding

Linear color space.

RENDER TARGETS

Render to a texture instead of screen:

let renderTarget = new THREE.WebGLRenderTarget(512, 512)

renderer.setRenderTarget(renderTarget)

```
renderer.render(scene, camera)
renderer.setRenderTarget(null)
```

Use the rendered texture:

```
let material = new THREE.MeshBasicMaterial({
 map: renderTarget.texture
})
```

MULTIPLE VIEWPORTS

Render different views:

```
renderer.setViewport(0, 0, halfWidth, height)
renderer.setScissor(0, 0, halfWidth, height)
renderer.setScissorTest(true)
renderer.render(scene, camera1)

renderer.setViewport(halfWidth, 0, halfWidth, height)
renderer.setScissor(halfWidth, 0, halfWidth, height)
renderer.render(scene, camera2)
```

setViewport defines where to render.
setScissor clips rendering to area.

PERFORMANCE INFO

Get rendering statistics:

```
console.log(renderer.info.render.calls)
console.log(renderer.info.render.triangles)
console.log(renderer.info.memory.geometries)
console.log(renderer.info.memory.textures)
```

render.calls - Draw calls per frame.
render.triangles - Triangles rendered.
memory.geometries - Geometries in memory.
memory.textures - Textures in memory.

CAPABILITIES

Check what the GPU supports:

```
let capabilities = renderer.capabilities

console.log(capabilities.maxTextures)
console.log(capabilities.maxTextureSize)
console.log(capabilities.maxCubemapSize)
console.log(capabilities.floatFragmentTextures)
```

DISPOSING

Clean up when done:

```
renderer.dispose()
```

Frees GPU resources.

Important when removing Three.js from page.

HANDLING RESIZE

Update renderer on window resize:

```
window.addEventListener("resize", function() {
 let width = window.innerWidth
 let height = window.innerHeight

 camera.aspect = width / height
 camera.updateProjectionMatrix()

 renderer.setSize(width, height)
})
```

SCREENSHOT

Capture the canvas:

```
renderer.render(scene, camera)

let screenshot = renderer.domElement.toDataURL("image/png")
```

Or trigger download:

```
function saveScreenshot() {
 renderer.render(scene, camera)

 let link = document.createElement("a")
 link.download = "screenshot.png"
 link.href = renderer.domElement.toDataURL("image/png")
 link.click()
}
```

Requires preserveDrawingBuffer: true.

PRACTICE

Create a renderer with antialiasing.

Set pixel ratio for sharp display.

Configure shadows.

Set up tone mapping.

Handle window resize.

WHAT YOU LEARNED

You learned to create and configure the renderer.

You learned about pixel ratio and size.

You learned shadow map settings.

You learned tone mapping and encoding.

You learned render targets and viewports.

The next chapter teaches geometry basics.
BOOK 18: THREE.JS COMPLETE
Chapter 5: Geometry Basics

WHAT IS GEOMETRY

Geometry defines the shape of an object.

It contains vertices (points) and faces (triangles).

Vertices define positions in 3D space.

Faces connect vertices into surfaces.

BOX GEOMETRY

A cube or rectangular box.

let geometry = new THREE.BoxGeometry(width, height, depth)

Simple cube:

let geometry = new THREE.BoxGeometry(1, 1, 1)

Rectangle:

let geometry = new THREE.BoxGeometry(2, 1, 0.5)

2 units wide, 1 unit tall, 0.5 units deep.

With segments for more detail:

let geometry = new THREE.BoxGeometry(1, 1, 1, 2, 2, 2)

Last three parameters are width, height, depth segments.

More segments allow smoother deformation.

SPHERE GEOMETRY

A ball shape.

```
let geometry = new THREE.SphereGeometry(radius,
widthSegments, heightSegments)
```

Basic sphere:

```
let geometry = new THREE.SphereGeometry(1, 32, 32)
```

Radius 1, 32 horizontal and vertical segments.

Low poly sphere:

```
let geometry = new THREE.SphereGeometry(1, 8, 6)
```

Partial sphere:

```
let geometry = new THREE.SphereGeometry(
 1,
 32,
 32,
 0,
 Math.PI * 2,
 0,
 Math.PI / 2
)
```

Last four parameters control start and length of horizontal and vertical sweep.

This creates a hemisphere.

PLANE GEOMETRY

A flat rectangle.

```
let geometry = new THREE.PlaneGeometry(width, height)
```

Basic plane:

```
let geometry = new THREE.PlaneGeometry(10, 10)
```

With segments:

```
let geometry = new THREE.PlaneGeometry(10, 10, 10, 10)
```

100 segments total.

Useful for terrain or deformation.

Planes face positive Z by default.

Rotate to use as ground:

```
plane.rotation.x = -Math.PI / 2
```

CYLINDER GEOMETRY

A tube shape.

```
let geometry = new THREE.CylinderGeometry(
 radiusTop,
 radiusBottom,
 height,
 radialSegments
)
```

Basic cylinder:

let geometry = new THREE.CylinderGeometry(1, 1, 2, 32)

Cone (zero top radius):

let geometry = new THREE.CylinderGeometry(0, 1, 2, 32)

Open ended:

let geometry = new THREE.CylinderGeometry(1, 1, 2, 32, 1, true)

Last parameter removes caps.

TORUS GEOMETRY

A donut shape.

let geometry = new THREE.TorusGeometry(radius, tube, radialSegments, tubularSegments)

Basic torus:

let geometry = new THREE.TorusGeometry(1, 0.4, 16, 100)

Radius 1, tube thickness 0.4.

TORUS KNOT GEOMETRY

A twisted donut.

let geometry = new THREE.TorusKnotGeometry(radius, tube,

tubularSegments, radialSegments, p, q)

Basic torus knot:

let geometry = new THREE.TorusKnotGeometry(1, 0.3, 100, 16)

Different knot types:

let geometry = new THREE.TorusKnotGeometry(1, 0.3, 100, 16, 2, 3)

p and q control the knot pattern.

CIRCLE GEOMETRY

A flat disc.

let geometry = new THREE.CircleGeometry(radius, segments)

Basic circle:

let geometry = new THREE.CircleGeometry(1, 32)

Partial circle:

let geometry = new THREE.CircleGeometry(1, 32, 0, Math.PI)

Last two parameters are start angle and arc length.

This creates a semicircle.

RING GEOMETRY

A flat ring or washer.

let geometry = new THREE.RingGeometry(innerRadius, outerRadius, segments)

Basic ring:

let geometry = new THREE.RingGeometry(0.5, 1, 32)

CONE GEOMETRY

A cone shape.

let geometry = new THREE.ConeGeometry(radius, height, radialSegments)

Basic cone:

let geometry = new THREE.ConeGeometry(1, 2, 32)

DODECAHEDRON GEOMETRY

A 12-sided shape.

let geometry = new THREE.DodecahedronGeometry(radius)

Basic:

let geometry = new THREE.DodecahedronGeometry(1)

ICOSAHEDRON GEOMETRY

A 20-sided shape.

let geometry = new THREE.IcosahedronGeometry(radius)

Basic:

let geometry = new THREE.IcosahedronGeometry(1)

With detail level:

let geometry = new THREE.IcosahedronGeometry(1, 2)

Higher detail subdivides faces for smoother sphere approximation.

OCTAHEDRON GEOMETRY

An 8-sided shape.

let geometry = new THREE.OctahedronGeometry(radius)

TETRAHEDRON GEOMETRY

A 4-sided pyramid.

let geometry = new THREE.TetrahedronGeometry(radius)

BUFFER GEOMETRY

All geometries in Three.js are BufferGeometry.

Data stored in typed arrays for GPU efficiency.

Access vertex data:

let positions = geometry.attributes.position.array

Returns Float32Array of x, y, z values.

Access normals:

let normals = geometry.attributes.normal.array

Access UVs:

let uvs = geometry.attributes.uv.array

MODIFYING GEOMETRY

Change vertex positions:

```
let positions = geometry.attributes.position.array

for (let i = 0; i < positions.length; i += 3) {
 positions[i] += Math.random() * 0.1
 positions[i + 1] += Math.random() * 0.1
 positions[i + 2] += Math.random() * 0.1
```

}

geometry.attributes.position.needsUpdate = true

needsUpdate tells Three.js to upload changes to GPU.

COMPUTING NORMALS

After modifying vertices recalculate normals:

geometry.computeVertexNormals()

Normals affect lighting.

BOUNDING BOX

Get the size of geometry:

```
geometry.computeBoundingBox()
let box = geometry.boundingBox

console.log(box.min)
console.log(box.max)
```

BOUNDING SPHERE

Get a sphere containing the geometry:

```
geometry.computeBoundingSphere()
let sphere = geometry.boundingSphere
```

```
console.log(sphere.center)
console.log(sphere.radius)
```

CENTERING GEOMETRY

Move geometry so center is at origin:

```
geometry.center()
```

DISPOSING GEOMETRY

Free memory when done:

```
geometry.dispose()
```

Important when removing objects.

COMBINING GEOMETRIES

Use BufferGeometryUtils:

```
import { mergeBufferGeometries } from "three/examples/jsm/utils/
BufferGeometryUtils.js"

let geometry1 = new THREE.BoxGeometry(1, 1, 1)
let geometry2 = new THREE.SphereGeometry(0.5, 32, 32)

geometry2.translate(2, 0, 0)

let merged = mergeBufferGeometries([geometry1, geometry2])
```

Merged geometry renders in one draw call.

PRACTICE

Create different geometry types.

Modify vertex positions.

Compute bounding boxes.

Center geometries.

Merge geometries.

WHAT YOU LEARNED

You learned built-in geometry types.

You learned geometry parameters.

You learned to access and modify vertices.

You learned about normals and bounds.

You learned to dispose and merge geometries.

The next chapter teaches materials.
BOOK 18: THREE.JS COMPLETE
Chapter 6: Materials Basics

WHAT IS A MATERIAL

A material defines how a surface looks.

It controls color, shininess, transparency, and texture.

Materials work with light to create the final appearance.

Different materials react to light differently.

MESHBASICMATERIAL

The simplest material.

Does not react to light.

Always shows its color regardless of scene lighting.

let material = new THREE.MeshBasicMaterial({ color: 0xff0000 })

Red material that ignores all lights.

Useful for:
- Debugging
- UI elements in 3D
- Stylized graphics
- Performance when lighting not needed

MESHLAMBERTMATERIAL

Reacts to light with diffuse shading.

Matte surface like unpolished wood or fabric.

```
let material = new THREE.MeshLambertMaterial({ color: 0x00ff00
})
```

Needs lights in scene to be visible.

Cheaper than Phong or Standard materials.

Good for non-shiny objects.

MESHPHONGMATERIAL

Reacts to light with specular highlights.

Shiny surface like plastic or polished wood.

```
let material = new THREE.MeshPhongMaterial({
 color: 0x0000ff,
 shininess: 100
})
```

shininess controls highlight sharpness.

Low values give soft highlights.

High values give sharp bright spots.

Specular color:

```
let material = new THREE.MeshPhongMaterial({
 color: 0x0000ff,
 specular: 0xffffff,
 shininess: 100
})
```

specular sets the color of highlights.

MESHSTANDARDMATERIAL

Physically based rendering material.

More realistic than Phong.

Uses roughness and metalness instead of shininess.

```
let material = new THREE.MeshStandardMaterial({
 color: 0xff6600,
 roughness: 0.5,
 metalness: 0.5
})
```

roughness - 0 is smooth mirror, 1 is completely rough.

metalness - 0 is non-metal, 1 is fully metallic.

This is the recommended material for realistic rendering.

MESHPHYSICALMATERIAL

Extended version of Standard material.

Adds clearcoat, sheen, and transmission.

```
let material = new THREE.MeshPhysicalMaterial({
 color: 0xffffff,
 roughness: 0.1,
```

```
 metalness: 0,
 clearcoat: 1.0,
 clearcoatRoughness: 0.1
})
```

Simulates car paint with clear coat layer.

Transmission for glass:

```
let material = new THREE.MeshPhysicalMaterial({
 color: 0xffffff,
 transmission: 1,
 thickness: 0.5,
 roughness: 0
})
```

Creates transparent glass effect.

MESHNORMALMATERIAL

Colors based on surface normals.

Each direction gets a different color.

```
let material = new THREE.MeshNormalMaterial()
```

Does not need lights.

Useful for debugging geometry.

MESHDEPTHMATERIAL

Colors based on distance from camera.

Near is white, far is black.

let material = new THREE.MeshDepthMaterial()

Used internally for shadow maps.

Can create fog-like effects.

MESHTOONMATERIAL

Cartoon style shading.

Sharp transitions between light and dark.

let material = new THREE.MeshToonMaterial({ color: 0xff0000 })

Creates cel-shaded look.

Custom gradient map for more control:

```
let gradientTexture = new THREE.DataTexture(
 new Uint8Array([0, 128, 255]),
 3, 1,
 THREE.RedFormat
)
gradientTexture.needsUpdate = true

let material = new THREE.MeshToonMaterial({
 color: 0xff0000,
 gradientMap: gradientTexture
})
```

MATERIAL PROPERTIES

Color:

material.color.set(0xff0000)
material.color.setHex(0xff0000)
material.color.setRGB(1, 0, 0)
material.color.setHSL(0, 1, 0.5)

Opacity:

material.transparent = true
material.opacity = 0.5

transparent must be true for opacity to work.

Side:

material.side = THREE.FrontSide
material.side = THREE.BackSide
material.side = THREE.DoubleSide

FrontSide renders only front faces (default).
BackSide renders only back faces.
DoubleSide renders both sides.

Wireframe:

material.wireframe = true

Shows geometry as lines instead of filled.

Flat shading:

material.flatShading = true

Gives faceted low-poly look.

COMBINING PROPERTIES

Create complex materials:

```
let material = new THREE.MeshStandardMaterial({
 color: 0x44aa88,
 roughness: 0.4,
 metalness: 0.1,
 transparent: true,
 opacity: 0.8,
 side: THREE.DoubleSide
})
```

MATERIAL ARRAYS

Different materials for different faces:

```
let materials = [
 new THREE.MeshBasicMaterial({ color: 0xff0000 }),
 new THREE.MeshBasicMaterial({ color: 0x00ff00 }),
 new THREE.MeshBasicMaterial({ color: 0x0000ff }),
 new THREE.MeshBasicMaterial({ color: 0xffff00 }),
 new THREE.MeshBasicMaterial({ color: 0xff00ff }),
 new THREE.MeshBasicMaterial({ color: 0x00ffff })
]

let cube = new THREE.Mesh(geometry, materials)
```

Each face of cube gets different color.

CLONING MATERIALS

Create a copy:

```
let material2 = material.clone()
material2.color.set(0x00ff00)
```

Changes to clone do not affect original.

DISPOSING MATERIALS

Free memory when done:

```
material.dispose()
```

Important when removing objects.

CHOOSING MATERIALS

Use MeshBasicMaterial when:
- No lighting needed
- Maximum performance needed
- Stylized flat look wanted

Use MeshLambertMaterial when:
- Need lighting but not highlights
- Performance matters
- Matte surfaces

Use MeshPhongMaterial when:
- Need shiny highlights
- Don't need physical accuracy

- Moderate performance needed

Use MeshStandardMaterial when:
- Want realistic rendering
- Working with PBR textures
- Quality matters more than performance

Use MeshPhysicalMaterial when:
- Need advanced effects
- Glass, car paint, fabric
- Maximum realism

PRACTICE

Create materials of each type.

Compare how they look with lights.

Adjust roughness and metalness.

Try transparent materials.

Use material arrays on a cube.

WHAT YOU LEARNED

You learned the different material types.

You learned material properties.

You learned when to use each material.

You learned about transparency and sides.

You learned to clone and dispose materials.

The next chapter teaches meshes.
BOOK 18: THREE.JS COMPLETE
Chapter 7: Meshes

WHAT IS A MESH

A mesh combines geometry and material.

Geometry defines the shape.

Material defines the appearance.

Mesh is the visible object in the scene.

```
let mesh = new THREE.Mesh(geometry, material)
```

CREATING A MESH

Basic mesh creation:

```
let geometry = new THREE.BoxGeometry(1, 1, 1)
let material = new THREE.MeshStandardMaterial({ color: 0x00ff00
})
let mesh = new THREE.Mesh(geometry, material)
scene.add(mesh)
```

MESH POSITION

Where the mesh is located:

```
mesh.position.x = 5
mesh.position.y = 2
mesh.position.z = -3
```

Or set all at once:

```
mesh.position.set(5, 2, -3)
```

Copy position from another object:

```
mesh.position.copy(otherMesh.position)
```

Add to current position:

```
mesh.position.add(new THREE.Vector3(1, 0, 0))
```

MESH ROTATION

How the mesh is rotated:

```
mesh.rotation.x = Math.PI / 4
mesh.rotation.y = Math.PI / 2
mesh.rotation.z = 0
```

Rotation uses radians.

Full rotation is Math.PI * 2.

Half rotation is Math.PI.

Quarter rotation is Math.PI / 2.

Convert degrees to radians:

let radians = THREE.MathUtils.degToRad(45)

Set rotation order:

mesh.rotation.order = "YXZ"

Default is "XYZ".

Order matters for combined rotations.

MESH SCALE

How big the mesh is:

mesh.scale.x = 2
mesh.scale.y = 0.5
mesh.scale.z = 1

Or set all at once:

mesh.scale.set(2, 0.5, 1)

Uniform scale:

mesh.scale.setScalar(2)

Makes mesh twice as big in all directions.

MESH VISIBILITY

Hide or show mesh:

mesh.visible = false

Mesh still exists but is not rendered.

mesh.visible = true

Shows mesh again.

MESH TRANSFORM METHODS

Look at a point:

```
mesh.lookAt(0, 0, 0)
mesh.lookAt(target.position)
```

Rotates mesh to face that point.

Translate along local axis:

```
mesh.translateX(5)
mesh.translateY(2)
mesh.translateZ(-3)
```

Moves in mesh local direction not world direction.

Translate on arbitrary axis:

```
let axis = new THREE.Vector3(1, 1, 0).normalize()
mesh.translateOnAxis(axis, 5)
```

Rotate around axis:

```
mesh.rotateX(0.1)
mesh.rotateY(0.1)
mesh.rotateZ(0.1)
```

Adds to current rotation.

Rotate on arbitrary axis:

```
let axis = new THREE.Vector3(0, 1, 0)
mesh.rotateOnAxis(axis, 0.1)
```

WORLD VS LOCAL TRANSFORMS

Local position is relative to parent.

World position is absolute in scene.

Get world position:

```
let worldPosition = new THREE.Vector3()
mesh.getWorldPosition(worldPosition)
```

Get world direction mesh is facing:

```
let direction = new THREE.Vector3()
mesh.getWorldDirection(direction)
```

MESH MATRICES

Transforms are stored in matrices.

mesh.matrix - Local transform matrix.

mesh.matrixWorld - World transform matrix.

Update matrices after manual changes:

```
mesh.updateMatrix()
mesh.updateMatrixWorld()
```

Disable auto matrix updates for performance:

```
mesh.matrixAutoUpdate = false
```

Then manually update when needed.

MESH USER DATA

Store custom data on mesh:

```
mesh.userData.health = 100
mesh.userData.type = "enemy"

let health = mesh.userData.health
```

MESH NAME

Give mesh a name:

```
mesh.name = "player"
```

Find by name:

```
let player = scene.getObjectByName("player")
```

CHANGING GEOMETRY

Replace geometry:

```
mesh.geometry.dispose()
mesh.geometry = new THREE.SphereGeometry(1, 32, 32)
```

CHANGING MATERIAL

Replace material:

```
mesh.material.dispose()
mesh.material = new THREE.MeshBasicMaterial({ color: 0xff0000
})
```

Access material properties:

```
mesh.material.color.set(0x00ff00)
mesh.material.opacity = 0.5
```

MESH LAYERS

Control which cameras see the mesh:

```
mesh.layers.set(1)
```

Camera must have that layer enabled to see it.

```
mesh.layers.enable(2)
mesh.layers.disable(1)
```

MESH FRUSTUM CULLING

Meshes outside camera view are not rendered.

```
mesh.frustumCulled = true
```

Default is true.

Set false for objects that should always render.

MESH RENDER ORDER

Control draw order for transparent objects:

```
mesh.renderOrder = 1
```

Higher values render later (on top).

MESH RAYCASTING

Meshes can be hit by raycasts.

```
let raycaster = new THREE.Raycaster()
raycaster.setFromCamera(mouse, camera)

let intersects = raycaster.intersectObject(mesh)
```

Returns array of intersections.

MESH SHADOWS

Enable shadows:

```
mesh.castShadow = true
mesh.receiveShadow = true
```

castShadow makes mesh create shadows.

receiveShadow makes mesh show shadows on it.

INSTANCED MESH

Many copies with different transforms:

```
let geometry = new THREE.BoxGeometry(1, 1, 1)
let material = new THREE.MeshStandardMaterial({ color: 0x00ff00
})
let count = 1000

let instancedMesh = new THREE.InstancedMesh(geometry,
material, count)
```

Set transform for each instance:

```
let matrix = new THREE.Matrix4()
let position = new THREE.Vector3()

for (let i = 0; i < count; i++) {
 position.set(
 Math.random() * 100 - 50,
```

```
 Math.random() * 100 - 50,
 Math.random() * 100 - 50
 )
 matrix.setPosition(position)
 instancedMesh.setMatrixAt(i, matrix)
}

instancedMesh.instanceMatrix.needsUpdate = true
```

Much faster than individual meshes.

SKINNED MESH

Mesh with bones for animation:

```
let skinnedMesh = gltfModel.scene.getObjectByProperty("type",
"SkinnedMesh")
```

Usually loaded from 3D model files.

Has skeleton with bones that deform the mesh.

LINE MESH

Line instead of filled shape:

```
let material = new THREE.LineBasicMaterial({ color: 0xffffff })
let points = []
points.push(new THREE.Vector3(0, 0, 0))
points.push(new THREE.Vector3(10, 0, 0))
points.push(new THREE.Vector3(10, 10, 0))

let geometry = new
THREE.BufferGeometry().setFromPoints(points)
```

```
let line = new THREE.Line(geometry, material)
```

POINTS MESH

Points instead of filled shape:

```
let material = new THREE.PointsMaterial({ color: 0xffffff, size: 0.1
})
let points = new THREE.Points(geometry, material)
```

Renders vertices as points.

DISPOSING MESH

Free memory:

```
mesh.geometry.dispose()
mesh.material.dispose()
scene.remove(mesh)
```

PRACTICE

Create meshes with different geometries.

Position rotate and scale meshes.

Use lookAt to orient meshes.

Create an instanced mesh.

Work with mesh userData.

WHAT YOU LEARNED

You learned to create and configure meshes.

You learned position rotation and scale.

You learned transform methods.

You learned about instanced meshes.

You learned mesh properties and disposal.

The next chapter teaches lighting.

BOOK 18: THREE.JS COMPLETE

Chapter 8: Lighting

WHY LIGHTING MATTERS

Without lights most materials appear black.

MeshBasicMaterial ignores lights.

MeshLambertMaterial needs lights to show color.

MeshStandardMaterial needs lights for realistic rendering.

Lighting creates depth, mood, and realism.

AMBIENT LIGHT

Illuminates all objects equally from all directions.

No shadows.

```
let light = new THREE.AmbientLight(0xffffff, 0.5)
scene.add(light)
```

First parameter is color.

Second is intensity.

Use for base illumination.

Prevents completely black shadows.

Usually combined with other lights.

DIRECTIONAL LIGHT

Parallel rays from a direction.

Like the sun.

```
let light = new THREE.DirectionalLight(0xffffff, 1)
light.position.set(5, 10, 7)
scene.add(light)
```

Position determines the direction rays come from.

Light shines toward the origin by default.

Change target:

```
light.target.position.set(0, 0, 0)
```

scene.add(light.target)

Or target an object:

light.target = someObject

Casts shadows.

POINT LIGHT

Emits light in all directions from a point.

Like a light bulb.

```
let light = new THREE.PointLight(0xffffff, 1, 100)
light.position.set(0, 10, 0)
scene.add(light)
```

Third parameter is distance.

Light intensity falls off with distance.

0 means no falloff (infinite range).

Decay controls falloff curve:

light.decay = 2

Default is 2 (physically correct).

Casts shadows.

SPOT LIGHT

Cone of light from a point.

Like a flashlight or stage light.

```
let light = new THREE.SpotLight(0xffffff, 1)
light.position.set(0, 10, 0)
scene.add(light)
```

Cone angle:

```
light.angle = Math.PI / 6
```

Smaller angle means tighter cone.

Penumbra (soft edges):

```
light.penumbra = 0.5
```

0 is sharp edge, 1 is completely soft.

Distance and decay:

```
light.distance = 100
light.decay = 2
```

Target:

```
light.target.position.set(0, 0, 0)
scene.add(light.target)
```

Casts shadows.

HEMISPHERE LIGHT

Sky and ground lighting.

Different color from above and below.

```
let light = new THREE.HemisphereLight(0x87ceeb, 0x362d1b, 1)
scene.add(light)
```

First parameter is sky color.

Second is ground color.

Third is intensity.

Creates natural outdoor lighting feel.

No shadows.

RECT AREA LIGHT

Rectangular light source.

Like a window or light panel.

```
let light = new THREE.RectAreaLight(0xffffff, 5, 4, 4)
light.position.set(0, 5, 0)
light.lookAt(0, 0, 0)
scene.add(light)
```

Parameters are color, intensity, width, height.

Requires RectAreaLightUniformsLib:

```
import { RectAreaLightUniformsLib } from "three/examples/jsm/
```

lights/RectAreaLightUniformsLib.js"

RectAreaLightUniformsLib.init()

Only works with MeshStandardMaterial and MeshPhysicalMaterial.

No shadows.

LIGHT COLOR

Set light color:

```
light.color.set(0xff0000)
light.color.setHex(0xff0000)
light.color.setRGB(1, 0, 0)
```

Warm light (sunset):

```
light.color.set(0xffaa33)
```

Cool light (overcast):

```
light.color.set(0xaaccff)
```

LIGHT INTENSITY

Control brightness:

```
light.intensity = 2
```

Higher values mean brighter light.

LIGHT HELPERS

Visualize lights for debugging.

Directional light helper:

```
let helper = new THREE.DirectionalLightHelper(light, 5)
scene.add(helper)
```

Point light helper:

```
let helper = new THREE.PointLightHelper(light, 1)
scene.add(helper)
```

Spot light helper:

```
let helper = new THREE.SpotLightHelper(light)
scene.add(helper)
```

Hemisphere light helper:

```
let helper = new THREE.HemisphereLightHelper(light, 5)
scene.add(helper)
```

Update helper after moving light:

```
helper.update()
```

MULTIPLE LIGHTS

Combine lights for realistic scenes:

```
let ambientLight = new THREE.AmbientLight(0x404040, 0.5)
scene.add(ambientLight)

let directionalLight = new THREE.DirectionalLight(0xffffff, 0.8)
directionalLight.position.set(10, 20, 10)
scene.add(directionalLight)

let pointLight = new THREE.PointLight(0xff6600, 1, 20)
pointLight.position.set(-5, 3, 0)
scene.add(pointLight)
```

PERFORMANCE CONSIDERATIONS

Each light costs performance.

Directional lights are cheapest.

Point and spot lights are more expensive.

Area lights are most expensive.

Limit number of lights.

Use ambient light to reduce need for many lights.

Bake lighting into textures for static scenes.

PHYSICALLY CORRECT LIGHTS

Enable physically accurate lighting:

renderer.physicallyCorrectLights = true

Light intensity uses real-world units (candela, lumen).

More realistic falloff.

LIGHT PROBES

Capture environment lighting:

```
let lightProbe = new THREE.LightProbe()
scene.add(lightProbe)
```

Usually generated from environment maps.

Provides ambient lighting from surroundings.

COMMON LIGHTING SETUPS

Three point lighting:

```
let keyLight = new THREE.DirectionalLight(0xffffff, 1)
keyLight.position.set(5, 5, 5)
scene.add(keyLight)

let fillLight = new THREE.DirectionalLight(0x8888ff, 0.5)
fillLight.position.set(-5, 0, 5)
scene.add(fillLight)

let backLight = new THREE.DirectionalLight(0xffffff, 0.3)
backLight.position.set(0, 5, -5)
scene.add(backLight)
```

Outdoor daylight:

```
let sunLight = new THREE.DirectionalLight(0xffffee, 1)
sunLight.position.set(50, 100, 50)
scene.add(sunLight)

let skyLight = new THREE.HemisphereLight(0x87ceeb, 0x362d1b,
0.5)
scene.add(skyLight)
```

Indoor scene:

```
let ambientLight = new THREE.AmbientLight(0x404040, 0.3)
scene.add(ambientLight)

let ceilingLight = new THREE.PointLight(0xffffee, 1, 20)
ceilingLight.position.set(0, 8, 0)
scene.add(ceilingLight)
```

ANIMATING LIGHTS

Move lights in animation loop:

```
function animate() {
 requestAnimationFrame(animate)

 let time = Date.now() * 0.001
 pointLight.position.x = Math.sin(time) * 5
 pointLight.position.z = Math.cos(time) * 5

 renderer.render(scene, camera)
}
```

PRACTICE

Add different light types to a scene.

Compare how materials look with different lights.

Use light helpers to see light positions.

Create a three point lighting setup.

Animate a light moving around the scene.

WHAT YOU LEARNED

You learned the different light types.

You learned light properties and colors.

You learned to use light helpers.

You learned common lighting setups.

You learned about light performance.

The next chapter teaches shadows.

BOOK 18: THREE.JS COMPLETE
Chapter 9: Shadows

WHY SHADOWS MATTER

Shadows ground objects in the scene.

Without shadows objects appear to float.

Shadows show spatial relationships.

They add depth and realism.

ENABLING SHADOWS

Three steps to enable shadows.

Step 1 - Enable on renderer:

renderer.shadowMap.enabled = true

Step 2 - Enable on light:

directionalLight.castShadow = true

Step 3 - Enable on objects:

cube.castShadow = true
floor.receiveShadow = true

castShadow means object creates shadows.

receiveShadow means shadows appear on object.

SHADOW MAP TYPES

Different shadow quality options.

renderer.shadowMap.type = THREE.BasicShadowMap

Fast but blocky shadows.

renderer.shadowMap.type = THREE.PCFShadowMap

Percentage closer filtering.

Smoother edges.

Default option.

renderer.shadowMap.type = THREE.PCFSoftShadowMap

Softer shadows.

More expensive.

renderer.shadowMap.type = THREE.VSMShadowMap

Variance shadow map.

Softest shadows.

Most expensive.

SHADOW MAP SIZE

Higher resolution means sharper shadows.

directionalLight.shadow.mapSize.width = 2048
directionalLight.shadow.mapSize.height = 2048

Default is 512.

Powers of 2 work best (512, 1024, 2048, 4096).

Higher values cost more performance and memory.

DIRECTIONAL LIGHT SHADOWS

Directional light uses orthographic shadow camera.

```
light.shadow.camera.left = -10
light.shadow.camera.right = 10
light.shadow.camera.top = 10
light.shadow.camera.bottom = -10
light.shadow.camera.near = 0.5
light.shadow.camera.far = 50
```

Visualize shadow camera:

```
let helper = new THREE.CameraHelper(light.shadow.camera)
scene.add(helper)
```

The shadow camera defines what area can have shadows.

Objects outside this area will not cast or receive shadows.

Make shadow camera tight around your scene for best quality.

POINT LIGHT SHADOWS

Point light shadows require rendering 6 times (cube map).

More expensive than directional.

pointLight.castShadow = true
pointLight.shadow.mapSize.width = 1024
pointLight.shadow.mapSize.height = 1024

Near and far planes:

pointLight.shadow.camera.near = 0.5
pointLight.shadow.camera.far = 50

SPOT LIGHT SHADOWS

Spot light uses perspective shadow camera.

spotLight.castShadow = true
spotLight.shadow.mapSize.width = 1024
spotLight.shadow.mapSize.height = 1024

Shadow camera settings:

spotLight.shadow.camera.near = 0.5
spotLight.shadow.camera.far = 50
spotLight.shadow.camera.fov = 30

SHADOW BIAS

Fixes shadow acne (striping artifacts).

light.shadow.bias = -0.0001

Small negative value usually fixes it.

Too much bias causes peter panning (shadow detaches from object).

Normal bias for thin objects:

light.shadow.normalBias = 0.02

SHADOW RADIUS

Softness for PCFSoftShadowMap:

light.shadow.radius = 4

Higher values make softer shadows.

Only works with PCFSoftShadowMap.

COMPLETE SHADOW SETUP

Full example:

```
renderer.shadowMap.enabled = true
renderer.shadowMap.type = THREE.PCFSoftShadowMap

let directionalLight = new THREE.DirectionalLight(0xffffff, 1)
directionalLight.position.set(5, 10, 7)
directionalLight.castShadow = true
directionalLight.shadow.mapSize.width = 2048
directionalLight.shadow.mapSize.height = 2048
directionalLight.shadow.camera.near = 0.5
directionalLight.shadow.camera.far = 50
directionalLight.shadow.camera.left = -10
```

```
directionalLight.shadow.camera.right = 10
directionalLight.shadow.camera.top = 10
directionalLight.shadow.camera.bottom = -10
directionalLight.shadow.bias = -0.0001
scene.add(directionalLight)

let cube = new THREE.Mesh(geometry, material)
cube.castShadow = true
scene.add(cube)

let floor = new THREE.Mesh(floorGeometry, floorMaterial)
floor.receiveShadow = true
scene.add(floor)
```

SHADOW CAMERA HELPER

Visualize the shadow camera frustum:

```
let shadowHelper = new
THREE.CameraHelper(light.shadow.camera)
scene.add(shadowHelper)
```

Shows the volume where shadows are calculated.

Helps tune shadow camera bounds.

SHADOW ONLY MATERIAL

Material that only shows shadows:

```
let material = new THREE.ShadowMaterial({ opacity: 0.5 })
let floor = new THREE.Mesh(floorGeometry, material)
floor.receiveShadow = true
```

Floor is invisible but shadows appear on it.

Useful for AR or compositing.

BAKED SHADOWS

For static scenes bake shadows into textures.

More performant than real-time shadows.

Create in 3D software and apply as texture.

```
let shadowTexture = textureLoader.load("baked-shadow.png")
let material = new THREE.MeshBasicMaterial({ map:
shadowTexture })
```

FAKE SHADOWS

Simple shadow plane under objects:

```
let shadowGeometry = new THREE.PlaneGeometry(1, 1)
let shadowMaterial = new THREE.MeshBasicMaterial({
 color: 0x000000,
 transparent: true,
 opacity: 0.3
})
let shadow = new THREE.Mesh(shadowGeometry,
shadowMaterial)
shadow.rotation.x = -Math.PI / 2
shadow.position.y = 0.01
```

Much cheaper than real shadows.

Move shadow with object.

SHADOW PERFORMANCE

Reduce shadow map size for distant lights.

Use one main shadow-casting light.

Keep shadow camera bounds tight.

Disable shadows on small or distant objects.

Use baked shadows for static geometry.

Consider fake shadows for simple cases.

CASCADE SHADOW MAPS

For large outdoor scenes use CSM.

Multiple shadow maps at different distances.

Requires addon:

```
import { CSM } from "three/examples/jsm/csm/CSM.js"

let csm = new CSM({
 maxFar: 1000,
 cascades: 4,
 shadowMapSize: 2048,
 lightDirection: new THREE.Vector3(-1, -1, -1).normalize(),
 camera: camera,
 parent: scene
})
```

Update each frame:

csm.update()

CONTACT SHADOWS

Soft shadows where objects meet ground.

import { ContactShadows } from "@react-three/drei"

Or implement manually with render-to-texture.

DEBUGGING SHADOWS

No shadows appearing:
- Check renderer.shadowMap.enabled
- Check light.castShadow
- Check mesh.castShadow and mesh.receiveShadow
- Check shadow camera bounds

Shadow cut off:
- Increase shadow camera far
- Expand shadow camera bounds

Shadow looks blocky:
- Increase shadow map size
- Use PCFSoftShadowMap

Shadow acne (stripes):
- Adjust shadow.bias

Shadow detached (peter panning):
- Reduce shadow.bias

PRACTICE

Enable shadows in your scene.

Try different shadow map types.

Adjust shadow camera bounds.

Use the shadow camera helper.

Create a floor with ShadowMaterial.

WHAT YOU LEARNED

You learned to enable shadows.

You learned shadow map types and sizes.

You learned to configure shadow cameras.

You learned about shadow bias.

You learned shadow optimization techniques.

The next chapter teaches textures.
BOOK 18: THREE.JS COMPLETE
Chapter 10: Textures

WHAT IS A TEXTURE

A texture is an image applied to a surface.

Adds detail without adding geometry.

Can define color, bumps, reflections, and more.

LOADING TEXTURES

Use TextureLoader:

```
let textureLoader = new THREE.TextureLoader()
let texture = textureLoader.load("wood.jpg")

let material = new THREE.MeshStandardMaterial({ map: texture })
```

With callback:

```
textureLoader.load(
 "wood.jpg",
 function(texture) {
 mesh.material.map = texture
 mesh.material.needsUpdate = true
 },
 function(progress) {
 console.log("Loading: " + progress.loaded + "/" + progress.total)
 },
 function(error) {
 console.error("Error loading texture")
 }
```

)

LOADING MANAGER

Track multiple textures:

```
let loadingManager = new THREE.LoadingManager()

loadingManager.onStart = function() {
 console.log("Started loading")
}

loadingManager.onProgress = function(url, loaded, total) {
 console.log("Loaded " + loaded + " of " + total)
}

loadingManager.onLoad = function() {
 console.log("All loaded")
}

loadingManager.onError = function(url) {
 console.error("Error loading " + url)
}

let textureLoader = new THREE.TextureLoader(loadingManager)
```

TEXTURE TYPES

Color map (diffuse):

```
material.map = texture
```

Main color/image of the surface.

Normal map:

```
material.normalMap = normalTexture
material.normalScale.set(1, 1)
```

Fakes bumps and detail without geometry.

Bump map:

```
material.bumpMap = bumpTexture
material.bumpScale = 0.05
```

Simpler version of normal map.

Grayscale image.

Displacement map:

```
material.displacementMap = displacementTexture
material.displacementScale = 0.5
```

Actually moves vertices.

Requires geometry with enough vertices.

Roughness map:

```
material.roughnessMap = roughnessTexture
```

Controls shininess across surface.

White is rough, black is smooth.

Metalness map:

material.metalnessMap = metalnessTexture

Controls metallic look across surface.

White is metal, black is non-metal.

Ambient occlusion map:

material.aoMap = aoTexture
material.aoMapIntensity = 1

Adds soft shadows in crevices.

Requires second UV set (uv2).

Emissive map:

material.emissiveMap = emissiveTexture
material.emissive.set(0xffffff)
material.emissiveIntensity = 1

Makes parts glow.

Alpha map:

material.alphaMap = alphaTexture
material.transparent = true

Controls transparency.

White is opaque, black is transparent.

Environment map:

material.envMap = envTexture

material.envMapIntensity = 1

Reflection of surroundings.

UV COORDINATES

UVs map texture positions to geometry.

U is horizontal (0 to 1).

V is vertical (0 to 1).

(0,0) is bottom left.

(1,1) is top right.

Built-in geometries have UVs already.

TEXTURE WRAPPING

Repeat mode:

texture.wrapS = THREE.RepeatWrapping
texture.wrapT = THREE.RepeatWrapping

Texture tiles when UV goes beyond 0-1.

S is horizontal, T is vertical.

Mirrored repeat:

texture.wrapS = THREE.MirroredRepeatWrapping

texture.wrapT = THREE.MirroredRepeatWrapping

Texture mirrors when tiling.

Clamp to edge:

texture.wrapS = THREE.ClampToEdgeWrapping
texture.wrapT = THREE.ClampToEdgeWrapping

Edge pixels stretch beyond bounds.

Default mode.

TEXTURE REPEAT

Tile the texture:

texture.wrapS = THREE.RepeatWrapping
texture.wrapT = THREE.RepeatWrapping
texture.repeat.set(4, 4)

Texture repeats 4 times in each direction.

TEXTURE OFFSET

Move the texture:

texture.offset.set(0.5, 0.5)

Values are in UV space (0 to 1).

TEXTURE ROTATION

Rotate the texture:

```
texture.rotation = Math.PI / 4
texture.center.set(0.5, 0.5)
```

Rotation is around the center point.

TEXTURE FILTERING

Magnification (texture smaller than surface):

texture.magFilter = THREE.LinearFilter

Smooth interpolation. Default.

texture.magFilter = THREE.NearestFilter

Pixelated look. Good for pixel art.

Minification (texture larger than surface):

texture.minFilter = THREE.LinearMipmapLinearFilter

Best quality. Default.

texture.minFilter = THREE.NearestFilter

Fastest but lowest quality.

MIPMAPS

Smaller versions of texture for distant objects.

Improves performance and quality.

Generated automatically by default.

Disable:

texture.generateMipmaps = false

ANISOTROPIC FILTERING

Improves texture quality at steep angles:

texture.anisotropy = renderer.capabilities.getMaxAnisotropy()

Higher values look better but cost performance.

TEXTURE ENCODING

Color textures should use sRGB:

texture.encoding = THREE.sRGBEncoding

Normal maps and other data textures stay linear:

normalTexture.encoding = THREE.LinearEncoding

CUBE TEXTURES

Six images for environment maps:

```
let cubeTextureLoader = new THREE.CubeTextureLoader()
let envMap = cubeTextureLoader.load([
 "px.jpg", "nx.jpg",
 "py.jpg", "ny.jpg",
 "pz.jpg", "nz.jpg"
])

scene.background = envMap
material.envMap = envMap
```

Order is positive X, negative X, positive Y, negative Y, positive Z, negative Z.

HDR ENVIRONMENT MAPS

High dynamic range for realistic lighting:

```
import { RGBELoader } from "three/examples/jsm/loaders/
RGBELoader.js"

let rgbeLoader = new RGBELoader()
rgbeLoader.load("environment.hdr", function(texture) {
 texture.mapping = THREE.EquirectangularReflectionMapping
 scene.background = texture
 scene.environment = texture
})
```

EQUIRECTANGULAR TEXTURES

Single 360 panorama image:

```
textureLoader.load("panorama.jpg", function(texture) {
 texture.mapping = THREE.EquirectangularReflectionMapping
 scene.background = texture
})
```

CANVAS TEXTURE

Create texture from canvas:

```
let canvas = document.createElement("canvas")
canvas.width = 256
canvas.height = 256
let ctx = canvas.getContext("2d")
ctx.fillStyle = "red"
ctx.fillRect(0, 0, 256, 256)

let texture = new THREE.CanvasTexture(canvas)
```

Update after drawing:

```
ctx.fillStyle = "blue"
ctx.fillRect(50, 50, 100, 100)
texture.needsUpdate = true
```

VIDEO TEXTURE

Create texture from video:

```
let video = document.createElement("video")
video.src = "video.mp4"
video.loop = true
video.muted = true
```

video.play()

let texture = new THREE.VideoTexture(video)

Updates automatically each frame.

DATA TEXTURE

Create texture from raw data:

```
let width = 256
let height = 256
let data = new Uint8Array(width * height * 4)

for (let i = 0; i < data.length; i += 4) {
 data[i] = Math.random() * 255
 data[i + 1] = Math.random() * 255
 data[i + 2] = Math.random() * 255
 data[i + 3] = 255
}

let texture = new THREE.DataTexture(data, width, height)
texture.needsUpdate = true
```

TEXTURE COMPRESSION

Compressed textures load faster and use less memory.

Formats like KTX2 with Basis Universal:

```
import { KTX2Loader } from "three/examples/jsm/loaders/
KTX2Loader.js"

let ktx2Loader = new KTX2Loader()
```

```
ktx2Loader.setTranscoderPath("basis/")
ktx2Loader.detectSupport(renderer)

ktx2Loader.load("texture.ktx2", function(texture) {
 material.map = texture
})
```

DISPOSING TEXTURES

Free memory when done:

```
texture.dispose()
```

Important when changing textures dynamically.

PBR TEXTURE WORKFLOW

Standard PBR texture set:

```
let textureLoader = new THREE.TextureLoader()

let colorTexture = textureLoader.load("color.jpg")
colorTexture.encoding = THREE.sRGBEncoding

let normalTexture = textureLoader.load("normal.jpg")
let roughnessTexture = textureLoader.load("roughness.jpg")
let metalnessTexture = textureLoader.load("metalness.jpg")
let aoTexture = textureLoader.load("ao.jpg")

let material = new THREE.MeshStandardMaterial({
 map: colorTexture,
 normalMap: normalTexture,
 roughnessMap: roughnessTexture,
 metalnessMap: metalnessTexture,
```

```
 aoMap: aoTexture
})
```

PRACTICE

Load textures and apply to materials.

Use normal maps to add detail.

Set up texture repeat and offset.

Create an environment map.

Make a PBR material with multiple textures.

WHAT YOU LEARNED

You learned to load and apply textures.

You learned texture types and their uses.

You learned texture wrapping and filtering.

You learned about environment maps.

You learned texture optimization techniques.

The next chapter teaches animation basics.
BOOK 18: THREE.JS COMPLETE
Chapter 11: Animation Basics

WHAT IS ANIMATION

Animation is change over time.

Moving objects, rotating cameras, changing colors.

Each frame shows a slightly different state.

At 60 frames per second motion appears smooth.

THE ANIMATION LOOP

Use requestAnimationFrame for smooth animation:

```
function animate() {
 requestAnimationFrame(animate)
 renderer.render(scene, camera)
}

animate()
```

requestAnimationFrame calls your function before next screen refresh.

Typically 60 times per second.

Pauses when tab is hidden (saves resources).

ANIMATING PROPERTIES

Change values each frame:

```
function animate() {
```

```
 requestAnimationFrame(animate)

 cube.rotation.x += 0.01
 cube.rotation.y += 0.01

 renderer.render(scene, camera)
}
```

Cube rotates continuously.

TIME BASED ANIMATION

Frame rate varies between devices.

Use time for consistent speed:

```
let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)

 let delta = clock.getDelta()
 cube.rotation.y += delta * 2

 renderer.render(scene, camera)
}
```

getDelta returns seconds since last call.

Rotation speed is now consistent regardless of frame rate.

ELAPSED TIME

Total time since start:

```
let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)

 let elapsed = clock.getElapsedTime()
 cube.position.x = Math.sin(elapsed) * 3
 cube.position.y = Math.cos(elapsed) * 3

 renderer.render(scene, camera)
}
```

Cube moves in a circle.

ANIMATION CURVES

Sine wave for smooth oscillation:

```
let elapsed = clock.getElapsedTime()
cube.position.y = Math.sin(elapsed * 2) * 2
```

Bounces between -2 and 2.

Cosine for offset wave:

```
cube.position.x = Math.cos(elapsed * 2) * 2
```

Combine for circular motion:

```
cube.position.x = Math.cos(elapsed) * 3
cube.position.z = Math.sin(elapsed) * 3
```

Pulse scale:

```
let scale = 1 + Math.sin(elapsed * 4) * 0.2
cube.scale.set(scale, scale, scale)
```

LERP FOR SMOOTH TRANSITIONS

Linear interpolation between values:

```
cube.position.x = THREE.MathUtils.lerp(cube.position.x, targetX,
0.1)
```

Moves 10% of remaining distance each frame.

Creates smooth easing motion.

Lerp for all properties:

```
cube.position.lerp(targetPosition, 0.1)
cube.quaternion.slerp(targetQuaternion, 0.1)
```

slerp is spherical lerp for rotations.

DAMPING

Smooth velocity reduction:

```
let velocity = new THREE.Vector3()
let damping = 0.95

function animate() {
 requestAnimationFrame(animate)
```

```
 cube.position.add(velocity)
 velocity.multiplyScalar(damping)

 renderer.render(scene, camera)
}
```

Velocity decreases each frame.

KEYFRAME ANIMATION

Three.js animation system uses keyframes.

Create animation clip:

```
let times = [0, 1, 2]
let values = [0, 0, 0, 5, 0, 0, 0, 0, 0]

let positionTrack = new THREE.VectorKeyframeTrack(
 ".position",
 times,
 values
)

let clip = new THREE.AnimationClip("move", 2, [positionTrack])
```

Play animation:

```
let mixer = new THREE.AnimationMixer(cube)
let action = mixer.clipAction(clip)
action.play()
```

Update mixer each frame:

```
let clock = new THREE.Clock()
```

```
function animate() {
 requestAnimationFrame(animate)

 let delta = clock.getDelta()
 mixer.update(delta)

 renderer.render(scene, camera)
}
```

ANIMATION ACTIONS

Control playback:

```
action.play()
action.stop()
action.pause()
action.reset()
```

Loop modes:

```
action.loop = THREE.LoopOnce
action.loop = THREE.LoopRepeat
action.loop = THREE.LoopPingPong
```

Playback speed:

```
action.timeScale = 2
```

Twice as fast.

```
action.timeScale = 0.5
```

Half speed.

Reverse:

action.timeScale = -1

Clamp when finished:

action.clampWhenFinished = true
action.loop = THREE.LoopOnce

Stays at final pose.

ANIMATION MIXING

Blend between animations:

let walkAction = mixer.clipAction(walkClip)
let runAction = mixer.clipAction(runClip)

walkAction.play()
runAction.play()

walkAction.weight = 0.7
runAction.weight = 0.3

70% walk, 30% run.

Crossfade between animations:

walkAction.crossFadeTo(runAction, 0.5, true)

Transitions over 0.5 seconds.

ANIMATION EVENTS

Listen for animation completion:

```
mixer.addEventListener("finished", function(event) {
 console.log("Animation finished: " + event.action.getClip().name)
})
```

Loop event:

```
mixer.addEventListener("loop", function(event) {
 console.log("Animation looped")
})
```

MORPH TARGETS

Blend between geometry shapes:

```
let geometry = new THREE.BoxGeometry(1, 1, 1)

let morphPositions = []
let sphereGeometry = new THREE.SphereGeometry(1, 32, 32)
morphPositions.push(sphereGeometry.attributes.position.array)

geometry.morphAttributes.position = [
 new THREE.Float32BufferAttribute(morphPositions[0], 3)
]

let mesh = new THREE.Mesh(geometry, material)
mesh.morphTargetInfluences[0] = 0.5
```

50% between box and sphere.

Animate morph:

```
function animate() {
 requestAnimationFrame(animate)

 let elapsed = clock.getElapsedTime()
 mesh.morphTargetInfluences[0] = (Math.sin(elapsed) + 1) / 2

 renderer.render(scene, camera)
}
```

SKELETAL ANIMATION

Bones deform mesh vertices.

Usually loaded from 3D models.

Access bones:

```
let skeleton = skinnedMesh.skeleton
let bone = skeleton.getBoneByName("arm")
```

Animate bone:

```
function animate() {
 requestAnimationFrame(animate)

 bone.rotation.z = Math.sin(clock.getElapsedTime()) * 0.5

 renderer.render(scene, camera)
}
```

GSAP FOR COMPLEX ANIMATIONS

Popular animation library works with Three.js:

npm install gsap

Animate position:

```
import gsap from "gsap"

gsap.to(cube.position, {
 x: 5,
 y: 2,
 duration: 2,
 ease: "power2.out"
})
```

Animate rotation:

```
gsap.to(cube.rotation, {
 y: Math.PI * 2,
 duration: 1,
 repeat: -1,
 ease: "none"
})
```

Timeline for sequenced animations:

```
let tl = gsap.timeline()

tl.to(cube.position, { x: 5, duration: 1 })
tl.to(cube.position, { y: 3, duration: 0.5 })
tl.to(cube.rotation, { z: Math.PI, duration: 0.5 })
```

TWEEN.JS ALTERNATIVE

Another animation library:

npm install @tweenjs/tween.js

```
import * as TWEEN from "@tweenjs/tween.js"

new TWEEN.Tween(cube.position)
 .to({ x: 5, y: 2 }, 2000)
 .easing(TWEEN.Easing.Quadratic.Out)
 .start()
```

Update in animation loop:

```
function animate() {
 requestAnimationFrame(animate)
 TWEEN.update()
 renderer.render(scene, camera)
}
```

SPRITE SHEET ANIMATION

Animate texture offset for 2D animations:

```
let frameWidth = 1 / 8
let currentFrame = 0

function animate() {
 requestAnimationFrame(animate)

 currentFrame = Math.floor(clock.getElapsedTime() * 10) % 8
 texture.offset.x = currentFrame * frameWidth

 renderer.render(scene, camera)
}
```

PRACTICE

Create a rotating cube with time-based animation.

Use lerp for smooth position changes.

Create a keyframe animation.

Use GSAP for complex animation sequence.

Animate material properties like color and opacity.

WHAT YOU LEARNED

You learned the animation loop.

You learned time-based animation.

You learned keyframe animation system.

You learned animation actions and mixing.

You learned external animation libraries.

The next chapter teaches OrbitControls.

BOOK 18: THREE.JS COMPLETE
Chapter 12: OrbitControls

WHAT ARE ORBIT CONTROLS

OrbitControls let the user rotate around a target.

Click and drag to orbit.

Scroll to zoom.

Right click to pan.

Essential for exploring 3D scenes.

IMPORTING ORBITCONTROLS

OrbitControls is in the examples folder:

```
import { OrbitControls } from "three/examples/jsm/controls/
OrbitControls.js"
```

BASIC SETUP

Create controls with camera and canvas:

```
let controls = new OrbitControls(camera, renderer.domElement)
```

That is it.

User can now orbit, zoom, and pan.

UPDATE IN ANIMATION LOOP

If using damping update each frame:

```
function animate() {
 requestAnimationFrame(animate)

 controls.update()

 renderer.render(scene, camera)
}
```

Required when enableDamping is true.

DAMPING

Smooth deceleration after releasing mouse:

```
controls.enableDamping = true
controls.dampingFactor = 0.05
```

Lower values mean slower deceleration.

Must call controls.update() each frame.

TARGET

Point the camera looks at:

```
controls.target.set(0, 1, 0)
controls.update()
```

Camera orbits around this point.

Look at an object:

```
controls.target.copy(mesh.position)
```

controls.update()

AUTO ROTATE

Automatically spin around target:

controls.autoRotate = true
controls.autoRotateSpeed = 2.0

Default speed is 2.0.

Positive rotates counterclockwise.

Negative rotates clockwise.

Must call controls.update() each frame.

ENABLE AND DISABLE

Enable or disable all controls:

controls.enabled = false

User cannot interact.

controls.enabled = true

User can interact again.

ROTATION LIMITS

Limit vertical rotation:

controls.minPolarAngle = 0
controls.maxPolarAngle = Math.PI / 2

Polar angle is vertical.

0 is looking straight down.

Math.PI is looking straight up.

Math.PI / 2 keeps camera above horizon.

Limit horizontal rotation:

controls.minAzimuthAngle = -Math.PI / 2
controls.maxAzimuthAngle = Math.PI / 2

Azimuth angle is horizontal.

Negative is left, positive is right.

Set both to same value to lock horizontal rotation.

ZOOM LIMITS

Limit zoom distance:

controls.minDistance = 2
controls.maxDistance = 20

For perspective camera.

For orthographic camera:

```
controls.minZoom = 0.5
controls.maxZoom = 4
```

ENABLE FEATURES

Turn features on or off:

```
controls.enableRotate = true
controls.enableZoom = true
controls.enablePan = true
```

Disable panning:

```
controls.enablePan = false
```

User cannot right-click drag.

Disable zoom:

```
controls.enableZoom = false
```

User cannot scroll to zoom.

PAN SETTINGS

Screen space panning:

```
controls.screenSpacePanning = true
```

Pan moves parallel to screen.

Default is true.

World space panning:

controls.screenSpacePanning = false

Pan moves along horizontal plane.

Limit pan area:

controls.maxTargetRadius = 10

Target cannot move beyond this distance from origin.

SPEED SETTINGS

Rotation speed:

controls.rotateSpeed = 1.0

Lower is slower.

Zoom speed:

controls.zoomSpeed = 1.0

Pan speed:

controls.panSpeed = 1.0

MOUSE BUTTONS

Change button assignments:

```
controls.mouseButtons = {
 LEFT: THREE.MOUSE.ROTATE,
 MIDDLE: THREE.MOUSE.DOLLY,
 RIGHT: THREE.MOUSE.PAN
}
```

Swap rotate and pan:

```
controls.mouseButtons = {
 LEFT: THREE.MOUSE.PAN,
 MIDDLE: THREE.MOUSE.DOLLY,
 RIGHT: THREE.MOUSE.ROTATE
}
```

TOUCH CONTROLS

Controls work on touch devices.

One finger rotates.

Two fingers zoom and pan.

Customize touch actions:

```
controls.touches = {
 ONE: THREE.TOUCH.ROTATE,
 TWO: THREE.TOUCH.DOLLY_PAN
}
```

KEYBOARD CONTROLS

Arrow keys pan by default:

```
controls.keys = {
 LEFT: "ArrowLeft",
 UP: "ArrowUp",
 RIGHT: "ArrowRight",
 BOTTOM: "ArrowDown"
}
```

Enable keyboard:

```
controls.listenToKeyEvents(window)
```

Disable keyboard:

```
controls.stopListenToKeyEvents()
```

SAVE AND RESTORE STATE

Save current state:

```
controls.saveState()
```

Reset to saved state:

```
controls.reset()
```

Returns to position when saveState was called.

EVENTS

Listen for changes:

```
controls.addEventListener("change", function() {
 console.log("Camera moved")
 renderer.render(scene, camera)
})
```

Start of interaction:

```
controls.addEventListener("start", function() {
 console.log("Interaction started")
})
```

End of interaction:

```
controls.addEventListener("end", function() {
 console.log("Interaction ended")
})
```

DISPOSING

Clean up when done:

```
controls.dispose()
```

OTHER CONTROL TYPES

FlyControls:

```
import { FlyControls } from "three/examples/jsm/controls/
FlyControls.js"

let controls = new FlyControls(camera, renderer.domElement)
controls.movementSpeed = 10
controls.rollSpeed = 0.5
```

WASD keys to move, mouse to look.

FirstPersonControls:

```
import { FirstPersonControls } from "three/examples/jsm/controls/
FirstPersonControls.js"

let controls = new FirstPersonControls(camera,
renderer.domElement)
controls.movementSpeed = 10
controls.lookSpeed = 0.1
```

Similar to fly but stays upright.

PointerLockControls:

```
import { PointerLockControls } from "three/examples/jsm/controls/
PointerLockControls.js"

let controls = new PointerLockControls(camera,
renderer.domElement)
scene.add(controls.getObject())

document.addEventListener("click", function() {
 controls.lock()
})
```

First person shooter style.

Mouse is captured and hidden.

TrackballControls:

```
import { TrackballControls } from "three/examples/jsm/controls/
TrackballControls.js"

let controls = new TrackballControls(camera,
renderer.domElement)
```

Like OrbitControls but can rotate freely without up vector.

DragControls:

```
import { DragControls } from "three/examples/jsm/controls/
DragControls.js"

let controls = new DragControls(objects, camera,
renderer.domElement)

controls.addEventListener("dragstart", function(event) {
 event.object.material.emissive.set(0xaaaaaa)
})

controls.addEventListener("dragend", function(event) {
 event.object.material.emissive.set(0x000000)
})
```

Drag objects in the scene.

TransformControls:

```
import { TransformControls } from "three/examples/jsm/controls/
TransformControls.js"

let controls = new TransformControls(camera,
renderer.domElement)
controls.attach(mesh)
scene.add(controls)
```

```
controls.setMode("translate")
controls.setMode("rotate")
controls.setMode("scale")
```

Gizmo for moving, rotating, scaling objects.

COMBINING CONTROLS

OrbitControls with TransformControls:

```
transformControls.addEventListener("dragging-changed",
function(event) {
 orbitControls.enabled = !event.value
})
```

Disable orbit when transforming.

PRACTICE

Add OrbitControls to your scene.

Set rotation and zoom limits.

Enable damping and auto rotate.

Try different control types.

Combine OrbitControls with DragControls.

WHAT YOU LEARNED

You learned to add OrbitControls.

You learned control limits and settings.

You learned events and state management.

You learned other control types.

You learned to combine controls.

The next chapter teaches loading 3D models.
BOOK 18: THREE.JS COMPLETE
Chapter 13: Loading 3D Models

WHY LOAD MODELS

Built-in geometries create simple shapes.

Real applications need complex models.

3D artists create models in Blender, Maya, or other software.

Three.js loads these models into your scene.

COMMON FORMATS

GLTF and GLB:

Recommended format for Three.js.

GLTF is JSON with separate binary files.

GLB is single binary file (easier to use).

Supports meshes, materials, textures, animations, lights.

Designed for web delivery.

OBJ:

Simple text format.

Only geometry and basic materials.

No animations.

Wide software support.

FBX:

Common in game industry.

Supports animations.

Larger file sizes.

DRACO:

Compressed geometry.

Much smaller files.

Requires decompression.

GLTF LOADER

Most important loader:

```
import { GLTFLoader } from "three/examples/jsm/loaders/
GLTFLoader.js"

let loader = new GLTFLoader()

loader.load(
 "model.glb",
 function(gltf) {
 scene.add(gltf.scene)
 },
 function(progress) {
 console.log((progress.loaded / progress.total * 100) + "% loaded")
 },
 function(error) {
 console.error("Error loading model", error)
 }
)
```

GLTF STRUCTURE

The loaded gltf object contains:

gltf.scene - The root object containing all meshes.

gltf.scenes - Array of scenes if multiple exist.

gltf.cameras - Cameras defined in the file.

gltf.animations - Animation clips.

gltf.asset - Metadata about the file.

WORKING WITH LOADED MODEL

Access the scene:

```
loader.load("model.glb", function(gltf) {
 let model = gltf.scene
 scene.add(model)
})
```

Position and scale:

```
loader.load("model.glb", function(gltf) {
 let model = gltf.scene
 model.position.set(0, 0, 0)
 model.scale.set(0.5, 0.5, 0.5)
 scene.add(model)
})
```

Find specific mesh:

```
loader.load("model.glb", function(gltf) {
 let model = gltf.scene

 model.traverse(function(child) {
 if (child.isMesh && child.name = = = "head") {
 child.material.color.set(0xff0000)
 }
 })

 scene.add(model)
})
```

ENABLING SHADOWS

Models need shadow settings:

```
loader.load("model.glb", function(gltf) {
 gltf.scene.traverse(function(child) {
 if (child.isMesh) {
 child.castShadow = true
 child.receiveShadow = true
 }
 })

 scene.add(gltf.scene)
})
```

PLAYING ANIMATIONS

Models may include animations:

```
let mixer

loader.load("model.glb", function(gltf) {
 scene.add(gltf.scene)

 mixer = new THREE.AnimationMixer(gltf.scene)

 gltf.animations.forEach(function(clip) {
 mixer.clipAction(clip).play()
 })
})
```

Update mixer in animation loop:

```
let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)
```

```
 let delta = clock.getDelta()

 if (mixer) {
 mixer.update(delta)
 }

 renderer.render(scene, camera)
}
```

DRACO COMPRESSION

Draco compresses geometry significantly.

Need Draco decoder:

```
import { GLTFLoader } from "three/examples/jsm/loaders/
GLTFLoader.js"
import { DRACOLoader } from "three/examples/jsm/loaders/
DRACOLoader.js"

let dracoLoader = new DRACOLoader()
dracoLoader.setDecoderPath("draco/")

let gltfLoader = new GLTFLoader()
gltfLoader.setDRACOLoader(dracoLoader)

gltfLoader.load("compressed-model.glb", function(gltf) {
 scene.add(gltf.scene)
})
```

Copy Draco decoder files to your project from:

node_modules/three/examples/jsm/libs/draco/

KTX2 TEXTURE COMPRESSION

Compressed textures in GLTF:

```
import { KTX2Loader } from "three/examples/jsm/loaders/
KTX2Loader.js"

let ktx2Loader = new KTX2Loader()
ktx2Loader.setTranscoderPath("basis/")
ktx2Loader.detectSupport(renderer)

gltfLoader.setKTX2Loader(ktx2Loader)
```

OBJ LOADER

For OBJ files:

```
import { OBJLoader } from "three/examples/jsm/loaders/
OBJLoader.js"

let loader = new OBJLoader()

loader.load("model.obj", function(object) {
 scene.add(object)
})
```

With MTL materials:

```
import { MTLLoader } from "three/examples/jsm/loaders/
MTLLoader.js"
import { OBJLoader } from "three/examples/jsm/loaders/
OBJLoader.js"

let mtlLoader = new MTLLoader()
```

```
mtlLoader.load("model.mtl", function(materials) {
 materials.preload()

 let objLoader = new OBJLoader()
 objLoader.setMaterials(materials)

 objLoader.load("model.obj", function(object) {
 scene.add(object)
 })
})
```

FBX LOADER

For FBX files:

```
import { FBXLoader } from "three/examples/jsm/loaders/
FBXLoader.js"

let loader = new FBXLoader()

loader.load("model.fbx", function(object) {
 scene.add(object)

 if (object.animations.length > 0) {
 mixer = new THREE.AnimationMixer(object)
 mixer.clipAction(object.animations[0]).play()
 }
})
```

LOADING MANAGER

Track multiple models:

```
let loadingManager = new THREE.LoadingManager()
```

```
loadingManager.onStart = function(url, loaded, total) {
 console.log("Started loading: " + url)
}

loadingManager.onProgress = function(url, loaded, total) {
 console.log("Loading: " + loaded + "/" + total)
}

loadingManager.onLoad = function() {
 console.log("All assets loaded")
}

loadingManager.onError = function(url) {
 console.error("Error loading: " + url)
}

let gltfLoader = new GLTFLoader(loadingManager)
```

CACHING LOADED MODELS

Avoid loading same model multiple times:

```
let modelCache = {}

function loadModel(url, callback) {
 if (modelCache[url]) {
 callback(modelCache[url].clone())
 return
 }

 loader.load(url, function(gltf) {
 modelCache[url] = gltf.scene
 callback(gltf.scene.clone())
 })
}
```

MODEL OPTIMIZATION

Reduce polygon count:

Use decimation in Blender.

Fewer triangles means better performance.

Optimize textures:

Use compressed textures (KTX2, basis).

Reduce texture resolution.

Use DRACO compression:

Smaller file size.

Faster download.

Merge meshes:

Fewer draw calls.

Export as single mesh if possible.

Remove unused data:

Delete hidden objects.

Remove unused animations.

FINDING FREE MODELS

Sketchfab:

Many free models available.

Download in GLTF format.

Poly Pizza:

Free low poly models.

Mixamo:

Free character models and animations.

Export as FBX then convert to GLTF.

Quaternius:

Free game-ready assets.

CONVERTING MODELS

Use Blender to convert between formats.

Import any format.

Export as GLTF or GLB.

Export settings in Blender:

Format - Choose glTF Binary (.glb)

Apply Modifiers - Check this

Include Animations - If needed

Compress - Enable Draco

TROUBLESHOOTING

Model appears black:

Check if lights are in scene.

Check material settings.

Model too big or small:

Adjust scale.

model.scale.set(0.01, 0.01, 0.01)

Model not visible:

Check position.

Camera might be inside the model.

Textures missing:

Check texture paths.

Textures should be relative to model file.

Animations not playing:

Check if animations exist in file.

Verify mixer.update() is called each frame.

PRACTICE

Download a GLTF model from Sketchfab.

Load it into your scene.

Adjust position, rotation, and scale.

Enable shadows on the model.

Play any included animations.

WHAT YOU LEARNED

You learned to load GLTF and GLB files.

You learned different loader types.

You learned to play model animations.

You learned compression techniques.

You learned model optimization.

The next chapter teaches raycasting.
BOOK 18: THREE.JS COMPLETE
Chapter 14: Raycasting

WHAT IS RAYCASTING

Raycasting shoots an invisible ray from a point.

The ray detects what it hits.

Used for mouse picking, collision detection, and AI vision.

Click on objects in 3D space using raycasting.

CREATING A RAYCASTER

Create the raycaster object:

```
let raycaster = new THREE.Raycaster()
```

RAYCASTER PROPERTIES

origin - Where ray starts.

direction - Where ray points.

near - Minimum distance to check.

far - Maximum distance to check.

Set near and far:

```
raycaster.near = 0
raycaster.far = 100
```

MOUSE PICKING

Detect which object the mouse is over.

Step 1 - Track mouse position:

```
let mouse = new THREE.Vector2()

window.addEventListener("mousemove", function(event) {
 mouse.x = (event.clientX / window.innerWidth) * 2 - 1
 mouse.y = -(event.clientY / window.innerHeight) * 2 + 1
})
```

Mouse coordinates converted to -1 to 1 range.

X goes from -1 (left) to 1 (right).

Y goes from -1 (bottom) to 1 (top).

Step 2 - Cast ray from camera through mouse:

```
raycaster.setFromCamera(mouse, camera)
```

Step 3 - Check for intersections:

```
let intersects = raycaster.intersectObjects(scene.children)

if (intersects.length > 0) {
```

```
 console.log("Hit: " + intersects[0].object.name)
}
```

INTERSECTION OBJECT

Each intersection contains:

object - The mesh that was hit.

point - 3D point where ray hit the object.

distance - Distance from ray origin to hit point.

face - The triangle face that was hit.

faceIndex - Index of the face.

uv - UV coordinates at hit point.

Example:

```
if (intersects.length > 0) {
 let hit = intersects[0]
 console.log("Object:", hit.object.name)
 console.log("Point:", hit.point)
 console.log("Distance:", hit.distance)
}
```

CLICK TO SELECT

Detect clicks on objects:

```
let selectedObject = null
```

```
window.addEventListener("click", function(event) {
 mouse.x = (event.clientX / window.innerWidth) * 2 - 1
 mouse.y = -(event.clientY / window.innerHeight) * 2 + 1

 raycaster.setFromCamera(mouse, camera)
 let intersects = raycaster.intersectObjects(scene.children)

 if (intersects.length > 0) {
 if (selectedObject) {
 selectedObject.material.color.set(0xffffff)
 }

 selectedObject = intersects[0].object
 selectedObject.material.color.set(0xff0000)
 }
})
```

HOVER EFFECTS

Change appearance on hover:

```
let hoveredObject = null

function animate() {
 requestAnimationFrame(animate)

 raycaster.setFromCamera(mouse, camera)
 let intersects = raycaster.intersectObjects(scene.children)

 if (hoveredObject) {
 hoveredObject.material.emissive.set(0x000000)
 hoveredObject = null
 }

 if (intersects.length > 0) {
 hoveredObject = intersects[0].object
```

```
 hoveredObject.material.emissive.set(0x333333)
 }

 renderer.render(scene, camera)
}
```

CHANGING CURSOR

Show pointer cursor when hovering clickable objects:

```
function animate() {
 requestAnimationFrame(animate)

 raycaster.setFromCamera(mouse, camera)
 let intersects = raycaster.intersectObjects(clickableObjects)

 if (intersects.length > 0) {
 document.body.style.cursor = "pointer"
 } else {
 document.body.style.cursor = "default"
 }

 renderer.render(scene, camera)
}
```

RECURSIVE INTERSECTION

Check children of objects:

```
let intersects = raycaster.intersectObjects(scene.children, true)
```

Second parameter enables recursive checking.

Required for groups and loaded models.

INTERSECT SINGLE OBJECT

Check one object:

```
let intersects = raycaster.intersectObject(mesh)
```

FILTERING OBJECTS

Only check specific objects:

```
let clickable = [cube, sphere, cylinder]
let intersects = raycaster.intersectObjects(clickable)
```

Or use layers:

```
raycaster.layers.set(1)
```

Only checks objects on layer 1.

RAYCASTING FROM ANY POINT

Not just from camera:

```
let origin = new THREE.Vector3(0, 5, 0)
let direction = new THREE.Vector3(0, -1, 0)

raycaster.set(origin, direction)

let intersects = raycaster.intersectObjects(scene.children)
```

Ray from (0, 5, 0) pointing down.

GROUND POSITION FROM MOUSE

Find where mouse ray hits ground plane:

```
let groundPlane = new THREE.Plane(new THREE.Vector3(0, 1, 0),
0)
let groundPosition = new THREE.Vector3()

window.addEventListener("click", function(event) {
 mouse.x = (event.clientX / window.innerWidth) * 2 - 1
 mouse.y = -(event.clientY / window.innerHeight) * 2 + 1

 raycaster.setFromCamera(mouse, camera)
 raycaster.ray.intersectPlane(groundPlane, groundPosition)

 console.log("Ground position:", groundPosition)
})
```

DRAG AND DROP

Drag object along ground:

```
let dragging = null
let groundPlane = new THREE.Plane(new THREE.Vector3(0, 1, 0),
0)
let intersection = new THREE.Vector3()

window.addEventListener("mousedown", function(event) {
 mouse.x = (event.clientX / window.innerWidth) * 2 - 1
 mouse.y = -(event.clientY / window.innerHeight) * 2 + 1
```

```
 raycaster.setFromCamera(mouse, camera)
 let intersects = raycaster.intersectObjects(draggableObjects)

 if (intersects.length > 0) {
 dragging = intersects[0].object
 controls.enabled = false
 }
})

window.addEventListener("mousemove", function(event) {
 mouse.x = (event.clientX / window.innerWidth) * 2 - 1
 mouse.y = -(event.clientY / window.innerHeight) * 2 + 1

 if (dragging) {
 raycaster.setFromCamera(mouse, camera)
 raycaster.ray.intersectPlane(groundPlane, intersection)
 dragging.position.x = intersection.x
 dragging.position.z = intersection.z
 }
})

window.addEventListener("mouseup", function() {
 dragging = null
 controls.enabled = true
})
```

COLLISION DETECTION

Check if object collides with others:

```
function checkCollision(object, direction) {
 let origin = object.position.clone()
 raycaster.set(origin, direction)

 let intersects = raycaster.intersectObjects(obstacles)

 if (intersects.length > 0 && intersects[0].distance < 1) {
```

```
 return true
 }

 return false
}
```

Usage:

```
let forward = new THREE.Vector3(0, 0, -1)
forward.applyQuaternion(player.quaternion)

if (!checkCollision(player, forward)) {
 player.translateZ(-0.1)
}
```

LINE OF SIGHT

Check if enemy can see player:

```
function canSee(from, to) {
 let direction = new THREE.Vector3()
 direction.subVectors(to, from).normalize()

 let distance = from.distanceTo(to)

 raycaster.set(from, direction)
 raycaster.far = distance

 let intersects = raycaster.intersectObjects(walls)

 return intersects.length === 0
}
```

Usage:

```
if (canSee(enemy.position, player.position)) {
 console.log("Enemy sees player")
}
```

RAYCASTING SPRITES

Sprites are checked differently:

```
let intersects = raycaster.intersectObjects(sprites)
```

Works automatically with sprite objects.

RAYCASTING LINES

Check if ray hits a line:

```
raycaster.params.Line.threshold = 0.1
```

Threshold is how close ray must be to count as hit.

RAYCASTING POINTS

Check if ray hits point cloud:

```
raycaster.params.Points.threshold = 0.1
```

Threshold is hit radius around each point.

GPU PICKING

For many objects GPU picking is faster.

Render each object with unique color.

Read pixel color under mouse.

Map color back to object.

More complex but scales better.

PERFORMANCE TIPS

Reduce objects to check:

Only include interactive objects in array.

Use bounding box first:

```
let box = new THREE.Box3().setFromObject(mesh)
if (raycaster.ray.intersectsBox(box)) {
 let intersects = raycaster.intersectObject(mesh)
}
```

Use layers:

Put interactive objects on separate layer.

Set raycaster to only check that layer.

Cache raycaster results:

Do not raycast every frame if not needed.

PRACTICE

Add click detection to objects.

Create hover effects.

Implement drag and drop.

Make a simple point-and-click game.

Use raycasting for collision detection.

WHAT YOU LEARNED

You learned to create raycasters.

You learned mouse picking.

You learned click and hover detection.

You learned ground intersection.

You learned collision and line of sight.

The next chapter teaches particles.
BOOK 18: THREE.JS COMPLETE
Chapter 15: Particles

WHAT ARE PARTICLES

Particles are many small objects.

Used for effects like fire, smoke, rain, snow, stars.

Each particle is a single point or sprite.

Thousands render efficiently as one object.

POINTS GEOMETRY

Create particles from geometry:

```
let geometry = new THREE.BufferGeometry()
let count = 5000

let positions = new Float32Array(count * 3)

for (let i = 0; i < count * 3; i++) {
 positions[i] = (Math.random() - 0.5) * 10
}

geometry.setAttribute("position", new
THREE.BufferAttribute(positions, 3))
```

Create points material:

```
let material = new THREE.PointsMaterial({
 size: 0.1,
 color: 0xffffff
})
```

Create points object:

```
let particles = new THREE.Points(geometry, material)
scene.add(particles)
```

POINTSMATERIAL PROPERTIES

size - Size of each particle.

```
material.size = 0.1
```

sizeAttenuation - Particles get smaller with distance.

```
material.sizeAttenuation = true
```

Default is true.

Set false for constant size (HUD elements).

color - Particle color.

```
material.color.set(0xff0000)
```

map - Texture for each particle.

```
let texture = textureLoader.load("particle.png")
material.map = texture
```

transparent and opacity:

```
material.transparent = true
material.opacity = 0.8
```

alphaMap - Texture for transparency.

material.alphaMap = alphaTexture
material.transparent = true

alphaTest - Discard pixels below threshold.

material.alphaTest = 0.01

Prevents black edges on particles.

depthWrite - Write to depth buffer.

material.depthWrite = false

Fixes sorting issues with transparent particles.

blending - How particles blend with background.

material.blending = THREE.AdditiveBlending

Additive makes overlapping particles brighter.

Good for fire and glow effects.

vertexColors - Use per-particle colors.

material.vertexColors = true

Requires color attribute on geometry.

PER-PARTICLE COLORS

Add color attribute:

```
let colors = new Float32Array(count * 3)

for (let i = 0; i < count; i++) {
 colors[i * 3] = Math.random()
 colors[i * 3 + 1] = Math.random()
 colors[i * 3 + 2] = Math.random()
}

geometry.setAttribute("color", new THREE.BufferAttribute(colors,
3))

material.vertexColors = true
```

ANIMATING PARTICLES

Update positions each frame:

```
let positions = particles.geometry.attributes.position.array

function animate() {
 requestAnimationFrame(animate)

 for (let i = 0; i < count; i++) {
 let i3 = i * 3

 positions[i3 + 1] -= 0.01

 if (positions[i3 + 1] < -5) {
 positions[i3 + 1] = 5
 }
 }

 particles.geometry.attributes.position.needsUpdate = true

 renderer.render(scene, camera)
}
```

Particles fall down and reset at top.

WAVE ANIMATION

Create wave pattern:

```
let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)

 let elapsed = clock.getElapsedTime()

 for (let i = 0; i < count; i++) {
 let i3 = i * 3
 let x = positions[i3]
 let z = positions[i3 + 2]

 positions[i3 + 1] = Math.sin(elapsed + x + z) * 0.5
 }

 particles.geometry.attributes.position.needsUpdate = true

 renderer.render(scene, camera)
}
```

PARTICLE TEXTURE

Create circular particle texture:

```
let canvas = document.createElement("canvas")
canvas.width = 64
canvas.height = 64
```

```
let ctx = canvas.getContext("2d")

let gradient = ctx.createRadialGradient(32, 32, 0, 32, 32, 32)
gradient.addColorStop(0, "rgba(255,255,255,1)")
gradient.addColorStop(1, "rgba(255,255,255,0)")

ctx.fillStyle = gradient
ctx.fillRect(0, 0, 64, 64)

let texture = new THREE.CanvasTexture(canvas)

let material = new THREE.PointsMaterial({
 size: 0.5,
 map: texture,
 transparent: true,
 depthWrite: false
})
```

PARTICLE FROM GEOMETRY

Particles at mesh vertices:

```
let sphereGeometry = new THREE.SphereGeometry(2, 32, 32)
let particleGeometry = new THREE.BufferGeometry()

particleGeometry.setAttribute(
 "position",
 sphereGeometry.attributes.position
)

let particles = new THREE.Points(particleGeometry, material)
```

PARTICLE SYSTEM CLASS

Organized particle system:

```
class ParticleSystem {
 constructor(count) {
 this.count = count
 this.velocities = []

 let positions = new Float32Array(count * 3)

 for (let i = 0; i < count; i++) {
 positions[i * 3] = 0
 positions[i * 3 + 1] = 0
 positions[i * 3 + 2] = 0

 this.velocities.push({
 x: (Math.random() - 0.5) * 0.1,
 y: Math.random() * 0.1,
 z: (Math.random() - 0.5) * 0.1
 })
 }

 this.geometry = new THREE.BufferGeometry()
 this.geometry.setAttribute(
 "position",
 new THREE.BufferAttribute(positions, 3)
 )

 this.material = new THREE.PointsMaterial({
 size: 0.1,
 color: 0xffaa00,
 transparent: true,
 depthWrite: false
 })

 this.points = new THREE.Points(this.geometry, this.material)
 }

 update() {
 let positions = this.geometry.attributes.position.array
```

```
 for (let i = 0; i < this.count; i++) {
 let i3 = i * 3

 positions[i3] += this.velocities[i].x
 positions[i3 + 1] += this.velocities[i].y
 positions[i3 + 2] += this.velocities[i].z

 this.velocities[i].y -= 0.001

 if (positions[i3 + 1] < 0) {
 positions[i3] = 0
 positions[i3 + 1] = 0
 positions[i3 + 2] = 0

 this.velocities[i].x = (Math.random() - 0.5) * 0.1
 this.velocities[i].y = Math.random() * 0.1
 this.velocities[i].z = (Math.random() - 0.5) * 0.1
 }
 }

 this.geometry.attributes.position.needsUpdate = true
 }
}
```

Usage:

```
let particleSystem = new ParticleSystem(1000)
scene.add(particleSystem.points)

function animate() {
 requestAnimationFrame(animate)
 particleSystem.update()
 renderer.render(scene, camera)
}
```

RAIN EFFECT

Falling rain particles:

```
let count = 10000
let positions = new Float32Array(count * 3)
let velocities = []

for (let i = 0; i < count; i++) {
 positions[i * 3] = (Math.random() - 0.5) * 50
 positions[i * 3 + 1] = Math.random() * 30
 positions[i * 3 + 2] = (Math.random() - 0.5) * 50

 velocities.push(0.1 + Math.random() * 0.1)
}

let geometry = new THREE.BufferGeometry()
geometry.setAttribute("position", new
THREE.BufferAttribute(positions, 3))

let material = new THREE.PointsMaterial({
 size: 0.1,
 color: 0xaaaaff,
 transparent: true,
 opacity: 0.6
})

let rain = new THREE.Points(geometry, material)
scene.add(rain)

function animateRain() {
 let pos = rain.geometry.attributes.position.array

 for (let i = 0; i < count; i++) {
 pos[i * 3 + 1] -= velocities[i]

 if (pos[i * 3 + 1] < 0) {
 pos[i * 3 + 1] = 30
 }
 }
```

```
 rain.geometry.attributes.position.needsUpdate = true
}
```

SNOW EFFECT

Slower falling with horizontal drift:

```
function animateSnow() {
 let pos = snow.geometry.attributes.position.array
 let time = clock.getElapsedTime()

 for (let i = 0; i < count; i++) {
 let i3 = i * 3

 pos[i3] += Math.sin(time + i) * 0.01
 pos[i3 + 1] -= 0.02
 pos[i3 + 2] += Math.cos(time + i) * 0.01

 if (pos[i3 + 1] < 0) {
 pos[i3 + 1] = 20
 }
 }

 snow.geometry.attributes.position.needsUpdate = true
}
```

STAR FIELD

Stars in space:

```
let count = 10000
let positions = new Float32Array(count * 3)
let colors = new Float32Array(count * 3)
let sizes = new Float32Array(count)
```

```
for (let i = 0; i < count; i++) {
 let theta = Math.random() * Math.PI * 2
 let phi = Math.acos(Math.random() * 2 - 1)
 let radius = 100 + Math.random() * 100

 positions[i * 3] = radius * Math.sin(phi) * Math.cos(theta)
 positions[i * 3 + 1] = radius * Math.sin(phi) * Math.sin(theta)
 positions[i * 3 + 2] = radius * Math.cos(phi)

 colors[i * 3] = 0.8 + Math.random() * 0.2
 colors[i * 3 + 1] = 0.8 + Math.random() * 0.2
 colors[i * 3 + 2] = 0.8 + Math.random() * 0.2

 sizes[i] = Math.random() * 2
}
```

INSTANCED PARTICLES

For complex particle shapes use InstancedMesh:

```
let geometry = new THREE.IcosahedronGeometry(0.05, 0)
let material = new THREE.MeshStandardMaterial({ color: 0xffaa00
})
let count = 1000

let instancedMesh = new THREE.InstancedMesh(geometry,
material, count)

let matrix = new THREE.Matrix4()
let position = new THREE.Vector3()

for (let i = 0; i < count; i++) {
 position.set(
 (Math.random() - 0.5) * 10,
 (Math.random() - 0.5) * 10,
 (Math.random() - 0.5) * 10
```

```
 )
 matrix.setPosition(position)
 instancedMesh.setMatrixAt(i, matrix)
}

scene.add(instancedMesh)
```

GPU PARTICLES

For best performance use shaders.

Update particle positions on GPU.

Covered in shaders chapter.

PRACTICE

Create a star field background.

Make rain or snow effect.

Build a particle fountain.

Add particle texture with soft edges.

Create fire effect with additive blending.

WHAT YOU LEARNED

You learned to create particle systems.

You learned PointsMaterial properties.

You learned to animate particles.

You learned common particle effects.

You learned per-particle colors and sizes.

The next chapter teaches post processing.

BOOK 18: THREE.JS COMPLETE

Chapter 16: Post Processing

WHAT IS POST PROCESSING

Post processing applies effects after rendering.

Blur, bloom, color correction, vignette.

Effects are applied to the rendered image.

Like Instagram filters for 3D.

EFFECTCOMPOSER

EffectComposer manages post processing pipeline:

```
import { EffectComposer } from "three/examples/jsm/
postprocessing/EffectComposer.js"

let composer = new EffectComposer(renderer)
```

RENDER PASS

First pass renders the scene:

```
import { RenderPass } from "three/examples/jsm/postprocessing/
RenderPass.js"

let renderPass = new RenderPass(scene, camera)
composer.addPass(renderPass)
```

USING COMPOSER

Replace renderer.render with composer.render:

```
function animate() {
 requestAnimationFrame(animate)
 composer.render()
}
```

BLOOM EFFECT

Makes bright areas glow:

```
import { UnrealBloomPass } from "three/examples/jsm/
postprocessing/UnrealBloomPass.js"

let bloomPass = new UnrealBloomPass(
 new THREE.Vector2(window.innerWidth, window.innerHeight),
 1.5,
 0.4,
 0.85
)
```

composer.addPass(bloomPass)

Parameters:

Resolution - Size of bloom texture.

Strength - Intensity of bloom (1.5).

Radius - Size of glow (0.4).

Threshold - Brightness level to start blooming (0.85).

Adjust bloom:

```
bloomPass.strength = 2.0
bloomPass.radius = 0.5
bloomPass.threshold = 0.8
```

DOT SCREEN

Half-tone newspaper effect:

```
import { DotScreenPass } from "three/examples/jsm/
postprocessing/DotScreenPass.js"

let dotScreenPass = new DotScreenPass()
composer.addPass(dotScreenPass)
```

Adjust pattern:

```
dotScreenPass.uniforms.scale.value = 4
```

GLITCH EFFECT

Digital distortion:

```
import { GlitchPass } from "three/examples/jsm/postprocessing/
GlitchPass.js"

let glitchPass = new GlitchPass()
composer.addPass(glitchPass)
```

Constant glitch:

```
glitchPass.goWild = true
```

FILM GRAIN

Cinematic film effect:

```
import { FilmPass } from "three/examples/jsm/postprocessing/
FilmPass.js"

let filmPass = new FilmPass(0.35, 0.025, 648, false)
composer.addPass(filmPass)
```

Parameters:

Noise intensity (0.35).

Scanline intensity (0.025).

Scanline count (648).

Grayscale (false).

BOKEH (DEPTH OF FIELD)

Blurs objects based on distance:

```
import { BokehPass } from "three/examples/jsm/postprocessing/
BokehPass.js"

let bokehPass = new BokehPass(scene, camera, {
 focus: 10.0,
 aperture: 0.025,
 maxblur: 0.01
})

composer.addPass(bokehPass)
```

Parameters:

focus - Distance that is in focus.

aperture - Smaller values mean more blur outside focus.

maxblur - Maximum blur amount.

SSAO (AMBIENT OCCLUSION)

Darkens crevices and corners:

```
import { SSAOPass } from "three/examples/jsm/postprocessing/
SSAOPass.js"

let ssaoPass = new SSAOPass(scene, camera, width, height)
ssaoPass.kernelRadius = 16
composer.addPass(ssaoPass)
```

Adjust parameters:

```
ssaoPass.kernelRadius = 8
ssaoPass.minDistance = 0.005
ssaoPass.maxDistance = 0.1
```

OUTLINE EFFECT

Highlights selected objects:

```
import { OutlinePass } from "three/examples/jsm/postprocessing/
OutlinePass.js"

let outlinePass = new OutlinePass(
 new THREE.Vector2(window.innerWidth, window.innerHeight),
 scene,
 camera
)

outlinePass.selectedObjects = [mesh1, mesh2]
outlinePass.edgeStrength = 3.0
outlinePass.edgeGlow = 1.0
outlinePass.edgeThickness = 1.0
outlinePass.visibleEdgeColor.set(0xffffff)

composer.addPass(outlinePass)
```

FXAA ANTIALIASING

Smooths edges:

```
import { ShaderPass } from "three/examples/jsm/postprocessing/
```

```
ShaderPass.js"
import { FXAAShader } from "three/examples/jsm/shaders/
FXAAShader.js"

let fxaaPass = new ShaderPass(FXAAShader)
fxaaPass.uniforms.resolution.value.set(
 1 / window.innerWidth,
 1 / window.innerHeight
)

composer.addPass(fxaaPass)
```

SMAA ANTIALIASING

Better quality antialiasing:

```
import { SMAAPass } from "three/examples/jsm/postprocessing/
SMAAPass.js"

let smaaPass = new SMAAPass(window.innerWidth,
window.innerHeight)
composer.addPass(smaaPass)
```

COLOR CORRECTION

Adjust colors:

```
import { ShaderPass } from "three/examples/jsm/postprocessing/
ShaderPass.js"
import { ColorCorrectionShader } from "three/examples/jsm/
shaders/ColorCorrectionShader.js"

let colorPass = new ShaderPass(ColorCorrectionShader)
colorPass.uniforms.powRGB.value.set(1.2, 1.2, 1.2)
```

```
colorPass.uniforms.mulRGB.value.set(1.1, 1.1, 1.1)

composer.addPass(colorPass)
```

GAMMA CORRECTION

Fix color space:

```
import { GammaCorrectionShader } from "three/examples/jsm/
shaders/GammaCorrectionShader.js"

let gammaPass = new ShaderPass(GammaCorrectionShader)
composer.addPass(gammaPass)
```

VIGNETTE

Darkens edges:

```
import { ShaderPass } from "three/examples/jsm/postprocessing/
ShaderPass.js"
import { VignetteShader } from "three/examples/jsm/shaders/
VignetteShader.js"

let vignettePass = new ShaderPass(VignetteShader)
vignettePass.uniforms.offset.value = 1.0
vignettePass.uniforms.darkness.value = 1.0

composer.addPass(vignettePass)
```

COMBINING EFFECTS

Multiple passes in sequence:

let renderPass = new RenderPass(scene, camera)
composer.addPass(renderPass)

let bloomPass = new UnrealBloomPass(resolution, 1.5, 0.4, 0.85)
composer.addPass(bloomPass)

let filmPass = new FilmPass(0.35, 0.025, 648, false)
composer.addPass(filmPass)

let fxaaPass = new ShaderPass(FXAAShader)
fxaaPass.uniforms.resolution.value.set(1 / width, 1 / height)
composer.addPass(fxaaPass)

Effects apply in order added.

RENDER TO SCREEN

Last pass must render to screen:

lastPass.renderToScreen = true

By default last added pass renders to screen.

SELECTIVE BLOOM

Bloom only certain objects:

Create two scenes - bloomed and non-bloomed.

Render each separately.

Combine with custom shader.

Simpler approach - use emissive materials:

```
let glowMaterial = new THREE.MeshStandardMaterial({
 color: 0x000000,
 emissive: 0xff0000,
 emissiveIntensity: 2
})
```

Set bloom threshold below emissive intensity.

HANDLING RESIZE

Update composer on resize:

```
window.addEventListener("resize", function() {
 let width = window.innerWidth
 let height = window.innerHeight

 camera.aspect = width / height
 camera.updateProjectionMatrix()

 renderer.setSize(width, height)
 composer.setSize(width, height)

 fxaaPass.uniforms.resolution.value.set(1 / width, 1 / height)
})
```

CUSTOM SHADER PASS

Create your own effect:

```
let customShader = {
```

```
 uniforms: {
 tDiffuse: { value: null },
 amount: { value: 1.0 }
 },
 vertexShader: `
 varying vec2 vUv;
 void main() {
 vUv = uv;
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
 `,
 fragmentShader: `
 uniform sampler2D tDiffuse;
 uniform float amount;
 varying vec2 vUv;
 void main() {
 vec4 color = texture2D(tDiffuse, vUv);
 color.rgb = 1.0 - color.rgb;
 gl_FragColor = color;
 }
 `
}

let customPass = new ShaderPass(customShader)
composer.addPass(customPass)
```

This inverts colors.

PERFORMANCE CONSIDERATIONS

Post processing costs performance.

Each pass renders full screen.

Reduce resolution:

composer.setPixelRatio(0.5)

Disable when not needed:

bloomPass.enabled = false

Use simpler effects on mobile.

COMPLETE SETUP

Full example:

```
import { EffectComposer } from "three/examples/jsm/
postprocessing/EffectComposer.js"
import { RenderPass } from "three/examples/jsm/postprocessing/
RenderPass.js"
import { UnrealBloomPass } from "three/examples/jsm/
postprocessing/UnrealBloomPass.js"
import { ShaderPass } from "three/examples/jsm/postprocessing/
ShaderPass.js"
import { FXAAShader } from "three/examples/jsm/shaders/
FXAAShader.js"

let composer = new EffectComposer(renderer)

let renderPass = new RenderPass(scene, camera)
composer.addPass(renderPass)

let bloomPass = new UnrealBloomPass(
 new THREE.Vector2(window.innerWidth, window.innerHeight),
 1.0,
 0.3,
 0.9
)
```

```
composer.addPass(bloomPass)

let fxaaPass = new ShaderPass(FXAAShader)
fxaaPass.uniforms.resolution.value.set(
 1 / window.innerWidth,
 1 / window.innerHeight
)
composer.addPass(fxaaPass)

function animate() {
 requestAnimationFrame(animate)
 composer.render()
}
```

PRACTICE

Set up EffectComposer.

Add bloom effect.

Combine multiple effects.

Create vignette effect.

Handle window resize.

WHAT YOU LEARNED

You learned to set up post processing.

You learned common effects.

You learned to combine multiple passes.

You learned custom shader passes.

You learned performance considerations.

The next chapter teaches audio.

BOOK 18: THREE.JS COMPLETE

Chapter 17: Audio

WHAT IS 3D AUDIO

Audio that responds to position in 3D space.

Sounds get louder as you approach.

Sounds pan left/right based on direction.

Creates immersive experiences.

AUDIO LISTENER

The listener is like your ears.

Attach to camera:

```
let listener = new THREE.AudioListener()
camera.add(listener)
```

Listener position determines what you hear.

AUDIO LOADER

Load audio files:

```
let audioLoader = new THREE.AudioLoader()

audioLoader.load("sound.mp3", function(buffer) {
 console.log("Audio loaded")
})
```

GLOBAL AUDIO

Plays same volume everywhere:

```
let listener = new THREE.AudioListener()
camera.add(listener)

let sound = new THREE.Audio(listener)

let audioLoader = new THREE.AudioLoader()
audioLoader.load("music.mp3", function(buffer) {
 sound.setBuffer(buffer)
 sound.setLoop(true)
 sound.setVolume(0.5)
 sound.play()
})
```

Good for background music.

POSITIONAL AUDIO

Volume based on distance:

```
let listener = new THREE.AudioListener()
camera.add(listener)
```

```
let sound = new THREE.PositionalAudio(listener)

let audioLoader = new THREE.AudioLoader()
audioLoader.load("engine.mp3", function(buffer) {
 sound.setBuffer(buffer)
 sound.setLoop(true)
 sound.setVolume(1)
 sound.setRefDistance(1)
 sound.play()
})

mesh.add(sound)
```

Sound attached to mesh.

Gets quieter as camera moves away.

POSITIONAL AUDIO PARAMETERS

setRefDistance - Distance at full volume.

```
sound.setRefDistance(1)
```

At this distance sound is at full volume.

setMaxDistance - Maximum distance to hear.

```
sound.setMaxDistance(100)
```

setRolloffFactor - How fast volume drops.

```
sound.setRolloffFactor(1)
```

Higher values mean faster dropoff.

setDistanceModel - How distance affects volume.

```
sound.setDistanceModel("linear")
sound.setDistanceModel("inverse")
sound.setDistanceModel("exponential")
```

AUDIO CONTROLS

Play and stop:

```
sound.play()
sound.stop()
sound.pause()
```

Check if playing:

```
if (sound.isPlaying) {
 sound.stop()
}
```

Volume:

```
sound.setVolume(0.5)
sound.getVolume()
```

Playback rate:

```
sound.setPlaybackRate(1.5)
```

Faster playback (also changes pitch).

Loop:

```
sound.setLoop(true)
sound.setLoop(false)
```

AUDIO ANALYSER

Visualize audio:

```
let analyser = new THREE.AudioAnalyser(sound, 32)
```

Get frequency data:

```
function animate() {
 requestAnimationFrame(animate)

 let data = analyser.getFrequencyData()
 console.log(data)

 let average = analyser.getAverageFrequency()
 mesh.scale.y = 1 + average / 256

 renderer.render(scene, camera)
}
```

data is array of frequency values.

average is single number for overall volume.

AUDIO VISUALIZATION

Scale objects based on audio:

```
let analyser = new THREE.AudioAnalyser(sound, 32)
let dataArray = analyser.data

function animate() {
 requestAnimationFrame(animate)

 analyser.getFrequencyData()

 for (let i = 0; i < bars.length; i++) {
 let value = dataArray[i] / 256
 bars[i].scale.y = 0.1 + value * 2
 }

 renderer.render(scene, camera)
}
```

AUDIO HELPER

Visualize positional audio:

```
let helper = new THREE.PositionalAudioHelper(sound, 1)
sound.add(helper)
```

Shows cone of sound direction.

DIRECTIONAL AUDIO

Sound with direction:

```
sound.setDirectionalCone(90, 180, 0.1)
```

Parameters:

Inner cone angle (full volume).

Outer cone angle (starts fading).

Outer volume (volume outside outer cone).

Point sound in direction:

```
sound.rotation.y = Math.PI / 2
```

MULTIPLE SOUNDS

Multiple sounds on same object:

```
let footstepSound = new THREE.PositionalAudio(listener)
let engineSound = new THREE.PositionalAudio(listener)

audioLoader.load("footstep.mp3", function(buffer) {
 footstepSound.setBuffer(buffer)
})

audioLoader.load("engine.mp3", function(buffer) {
 engineSound.setBuffer(buffer)
 engineSound.setLoop(true)
 engineSound.play()
})

mesh.add(footstepSound)
mesh.add(engineSound)
```

SOUND POOL

Reuse sounds for efficiency:

```
class SoundPool {
 constructor(listener, buffer, count) {
 this.sounds = []
 this.index = 0

 for (let i = 0; i < count; i++) {
 let sound = new THREE.PositionalAudio(listener)
 sound.setBuffer(buffer)
 this.sounds.push(sound)
 }
 }

 play(position) {
 let sound = this.sounds[this.index]
 sound.position.copy(position)

 if (sound.isPlaying) {
 sound.stop()
 }

 sound.play()
 this.index = (this.index + 1) % this.sounds.length
 }
}
```

Usage:

```
let explosionPool

audioLoader.load("explosion.mp3", function(buffer) {
 explosionPool = new SoundPool(listener, buffer, 5)
})

function explode(position) {
 explosionPool.play(position)
}
```

USER INTERACTION REQUIREMENT

Browsers block audio until user interaction.

Start audio on click:

```
document.addEventListener("click", function() {
 if (sound.context.state === "suspended") {
 sound.context.resume()
 }
})
```

Or show play button:

```
let started = false

document.getElementById("startButton").addEventListener("click",
function() {
 if (!started) {
 sound.play()
 started = true
 this.style.display = "none"
 }
})
```

LOADING MULTIPLE SOUNDS

Use loading manager:

```
let audioLoader = new THREE.AudioLoader(loadingManager)

let sounds = {}

audioLoader.load("music.mp3", function(buffer) {
 sounds.music = buffer
```

```
})

audioLoader.load("explosion.mp3", function(buffer) {
 sounds.explosion = buffer
})

loadingManager.onLoad = function() {
 console.log("All sounds loaded")
 startGame()
}
```

WEB AUDIO API ACCESS

Access underlying Web Audio:

```
let audioContext = listener.context

let oscillator = audioContext.createOscillator()
oscillator.frequency.value = 440
oscillator.connect(listener.gain)
oscillator.start()
```

Full Web Audio API available.

REVERB AND EFFECTS

Use Web Audio for effects:

```
let convolver = audioContext.createConvolver()

fetch("reverb-impulse.wav")
 .then(response => response.arrayBuffer())
 .then(data => audioContext.decodeAudioData(data))
 .then(buffer => {
```

```
 convolver.buffer = buffer
 })

sound.setFilter(convolver)
```

AUDIO FORMATS

MP3 - Wide support, lossy compression.

OGG - Good quality, not all browsers.

WAV - Uncompressed, large files.

AAC - Good compression, wide support.

Provide multiple formats:

```
audioLoader.load(
 canPlayOgg ? "sound.ogg" : "sound.mp3",
 function(buffer) {
 sound.setBuffer(buffer)
 }
)
```

PRACTICE

Add background music to scene.

Create positional audio on an object.

Visualize audio with analyser.

Handle user interaction for autoplay.

Create a sound pool for repeated sounds.

WHAT YOU LEARNED

You learned global and positional audio.

You learned audio controls and parameters.

You learned audio visualization.

You learned sound pools.

You learned browser autoplay restrictions.

The next chapter teaches physics introduction.

BOOK 18: THREE.JS COMPLETE

Chapter 18: Physics Introduction

WHY PHYSICS

Physics makes objects behave realistically.

Objects fall with gravity.

Objects collide and bounce.

Objects have mass and momentum.

Three.js does not include physics.

Need external physics library.

PHYSICS LIBRARIES

Cannon.js:

Lightweight and easy to learn.

Good for simple physics.

No longer maintained but stable.

Cannon-es:

Fork of Cannon.js.

Actively maintained.

ES module support.

Ammo.js:

Port of Bullet physics engine.

More features and accuracy.

More complex to use.

Larger file size.

Rapier:

Modern and fast.

Written in Rust, compiled to WebAssembly.

Good performance.

CANNON-ES SETUP

Install:

npm install cannon-es

Import:

import * as CANNON from "cannon-es"

Create physics world:

let world = new CANNON.World()
world.gravity.set(0, -9.82, 0)

Gravity points down at 9.82 m/s squared.

PHYSICS BODIES

Physics world has bodies.

Three.js scene has meshes.

Keep them synchronized.

Create body:

let shape = new CANNON.Box(new CANNON.Vec3(0.5, 0.5, 0.5))

```
let body = new CANNON.Body({
 mass: 1,
 shape: shape
})
body.position.set(0, 5, 0)
world.addBody(body)
```

mass 0 means static (does not move).

mass greater than 0 means dynamic (affected by forces).

SHAPES

Box:

```
let shape = new CANNON.Box(new CANNON.Vec3(width/2,
height/2, depth/2))
```

Parameters are half-extents (half the size).

Sphere:

```
let shape = new CANNON.Sphere(radius)
```

Cylinder:

```
let shape = new CANNON.Cylinder(radiusTop, radiusBottom,
height, segments)
```

Plane:

```
let shape = new CANNON.Plane()
```

Infinite plane.

Rotate body to orient.

Convex hull:

```
let vertices = [
 new CANNON.Vec3(0, 0, 0),
 new CANNON.Vec3(1, 0, 0),
 new CANNON.Vec3(0, 1, 0),
 new CANNON.Vec3(0, 0, 1)
]
let shape = new CANNON.ConvexPolyhedron({ vertices: vertices })
```

Trimesh (from geometry):

```
let vertices = geometry.attributes.position.array
let indices = geometry.index.array
let shape = new CANNON.Trimesh(vertices, indices)
```

Only for static bodies.

SYNCHRONIZING THREE AND CANNON

Update Three.js meshes from physics bodies:

```
function animate() {
 requestAnimationFrame(animate)

 world.step(1/60)

 mesh.position.copy(body.position)
 mesh.quaternion.copy(body.quaternion)

 renderer.render(scene, camera)
```

```
}
```

FIXED TIMESTEP

Physics needs consistent time steps:

```
let clock = new THREE.Clock()
let delta
let accumulator = 0
let fixedTimeStep = 1/60

function animate() {
 requestAnimationFrame(animate)

 delta = clock.getDelta()
 accumulator += delta

 while (accumulator >= fixedTimeStep) {
 world.step(fixedTimeStep)
 accumulator -= fixedTimeStep
 }

 mesh.position.copy(body.position)
 mesh.quaternion.copy(body.quaternion)

 renderer.render(scene, camera)
}
```

COMPLETE FALLING BOX EXAMPLE

```
let scene = new THREE.Scene()
let camera = new THREE.PerspectiveCamera(75, width/height,
0.1, 1000)
camera.position.set(0, 5, 10)
```

```
camera.lookAt(0, 0, 0)

let renderer = new THREE.WebGLRenderer()
renderer.setSize(width, height)
document.body.appendChild(renderer.domElement)

let world = new CANNON.World()
world.gravity.set(0, -9.82, 0)

let groundShape = new CANNON.Plane()
let groundBody = new CANNON.Body({ mass: 0, shape:
groundShape })
groundBody.quaternion.setFromAxisAngle(new CANNON.Vec3(1, 0,
0), -Math.PI/2)
world.addBody(groundBody)

let groundGeometry = new THREE.PlaneGeometry(20, 20)
let groundMaterial = new THREE.MeshStandardMaterial({ color:
0x808080 })
let groundMesh = new THREE.Mesh(groundGeometry,
groundMaterial)
groundMesh.rotation.x = -Math.PI/2
scene.add(groundMesh)

let boxShape = new CANNON.Box(new CANNON.Vec3(0.5, 0.5,
0.5))
let boxBody = new CANNON.Body({ mass: 1, shape: boxShape })
boxBody.position.set(0, 5, 0)
world.addBody(boxBody)

let boxGeometry = new THREE.BoxGeometry(1, 1, 1)
let boxMaterial = new THREE.MeshStandardMaterial({ color:
0xff0000 })
let boxMesh = new THREE.Mesh(boxGeometry, boxMaterial)
scene.add(boxMesh)

let light = new THREE.DirectionalLight(0xffffff, 1)
light.position.set(5, 10, 5)
scene.add(light)
scene.add(new THREE.AmbientLight(0x404040))
```

```
function animate() {
 requestAnimationFrame(animate)

 world.step(1/60)

 boxMesh.position.copy(boxBody.position)
 boxMesh.quaternion.copy(boxBody.quaternion)

 renderer.render(scene, camera)
}

animate()
```

VELOCITY

Set body velocity:

```
body.velocity.set(5, 10, 0)
```

Get velocity:

```
console.log(body.velocity)
```

APPLYING FORCES

Apply force at center:

```
body.applyForce(new CANNON.Vec3(100, 0, 0))
```

Apply force at point:

```
body.applyForce(
 new CANNON.Vec3(100, 0, 0),
 new CANNON.Vec3(0, 0.5, 0)
)
```

Second parameter is world point.

Apply impulse (instant force):

```
body.applyImpulse(new CANNON.Vec3(0, 10, 0))
```

MATERIALS AND FRICTION

Create physics material:

```
let groundMaterial = new CANNON.Material("ground")
let boxMaterial = new CANNON.Material("box")
```

Create contact material:

```
let contactMaterial = new CANNON.ContactMaterial(
 groundMaterial,
 boxMaterial,
 {
 friction: 0.4,
 restitution: 0.6
 }
)

world.addContactMaterial(contactMaterial)
```

friction controls sliding.

restitution controls bounciness.

Assign materials:

```
groundBody.material = groundMaterial
boxBody.material = boxMaterial
```

COLLISION EVENTS

Detect collisions:

```
body.addEventListener("collide", function(event) {
 console.log("Collision with", event.body)
 console.log("Impact velocity",
event.contact.getImpactVelocityAlongNormal())
})
```

RAYCASTING IN PHYSICS

Cast ray in physics world:

```
let from = new CANNON.Vec3(0, 10, 0)
let to = new CANNON.Vec3(0, -10, 0)

let result = new CANNON.RaycastResult()
world.raycastClosest(from, to, {}, result)

if (result.hasHit) {
 console.log("Hit at", result.hitPointWorld)
 console.log("Hit body", result.body)
}
```

CONSTRAINTS

Point to point constraint:

```
let constraint = new CANNON.PointToPointConstraint(
 bodyA,
 new CANNON.Vec3(0, 0.5, 0),
 bodyB,
 new CANNON.Vec3(0, -0.5, 0)
)
world.addConstraint(constraint)
```

Connects two bodies at points.

Hinge constraint:

```
let constraint = new CANNON.HingeConstraint(
 bodyA,
 bodyB,
 {
 pivotA: new CANNON.Vec3(0, 0.5, 0),
 pivotB: new CANNON.Vec3(0, -0.5, 0),
 axisA: new CANNON.Vec3(0, 0, 1),
 axisB: new CANNON.Vec3(0, 0, 1)
 }
)
world.addConstraint(constraint)
```

Lock constraint:

```
let constraint = new CANNON.LockConstraint(bodyA, bodyB)
world.addConstraint(constraint)
```

Bodies move together as one.

REMOVING BODIES

Remove body from world:

world.removeBody(body)

PERFORMANCE TIPS

Use simple shapes:

Boxes and spheres are fast.

Meshes are slow.

Reduce body count:

Combine static geometry.

Remove distant bodies.

Adjust solver iterations:

world.solver.iterations = 10

Lower is faster but less accurate.

Use sleep:

world.allowSleep = true
body.allowSleep = true
body.sleepSpeedLimit = 0.1
body.sleepTimeLimit = 1

Sleeping bodies stop being simulated.

HELPER CLASS

Manage physics objects:

```
class PhysicsObject {
 constructor(mesh, body) {
 this.mesh = mesh
 this.body = body
 }

 update() {
 this.mesh.position.copy(this.body.position)
 this.mesh.quaternion.copy(this.body.quaternion)
 }
}

let objects = []

function createBox(x, y, z) {
 let geometry = new THREE.BoxGeometry(1, 1, 1)
 let material = new THREE.MeshStandardMaterial({ color:
0xff0000 })
 let mesh = new THREE.Mesh(geometry, material)
 scene.add(mesh)

 let shape = new CANNON.Box(new CANNON.Vec3(0.5, 0.5, 0.5))
 let body = new CANNON.Body({ mass: 1, shape: shape })
 body.position.set(x, y, z)
 world.addBody(body)

 objects.push(new PhysicsObject(mesh, body))
}

function animate() {
```

```
 requestAnimationFrame(animate)

 world.step(1/60)

 for (let obj of objects) {
 obj.update()
 }

 renderer.render(scene, camera)
}
```

PRACTICE

Create a falling box with ground.

Add multiple falling objects.

Implement bounciness with restitution.

Detect collisions.

Create a simple physics playground.

WHAT YOU LEARNED

You learned physics library options.

You learned bodies and shapes.

You learned synchronizing physics with Three.js.

You learned forces and materials.

You learned constraints and events.

The next chapter teaches shaders introduction.
BOOK 18: THREE.JS COMPLETE
Chapter 19: Shaders Introduction

WHAT ARE SHADERS

Shaders are programs that run on the GPU.

They determine how things look.

Vertex shaders position vertices.

Fragment shaders color pixels.

Written in GLSL (OpenGL Shading Language).

WHY USE CUSTOM SHADERS

Built-in materials are limited.

Custom shaders create unique effects.

Water, fire, portals, force fields.

Animated textures and patterns.

Better performance for specific effects.

SHADERMATERIAL

Create material with custom shaders:

```
let material = new THREE.ShaderMaterial({
 vertexShader: vertexShaderCode,
 fragmentShader: fragmentShaderCode
})
```

BASIC VERTEX SHADER

Positions vertices on screen:

```
let vertexShader = `
 void main() {
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
`
```

projectionMatrix - Camera projection.

modelViewMatrix - Object transform plus camera view.

position - Vertex position (provided by Three.js).

gl_Position - Output position on screen.

BASIC FRAGMENT SHADER

Colors each pixel:

```
let fragmentShader = `
 void main() {
 gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
```

```
}
`
```

gl_FragColor is the output color.

vec4 is red, green, blue, alpha (0 to 1).

This colors everything red.

COMPLETE EXAMPLE

Red shader material:

```
let material = new THREE.ShaderMaterial({
 vertexShader: `
 void main() {
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
 `,
 fragmentShader: `
 void main() {
 gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
 }
`
})

let mesh = new THREE.Mesh(geometry, material)
```

UNIFORMS

Pass data from JavaScript to shader:

```
let material = new THREE.ShaderMaterial({
```

```
 uniforms: {
 uTime: { value: 0.0 },
 uColor: { value: new THREE.Color(0xff0000) }
 },
 vertexShader: vertexShader,
 fragmentShader: fragmentShader
})
```

Use in shader:

```
let fragmentShader = `
 uniform float uTime;
 uniform vec3 uColor;

 void main() {
 gl_FragColor = vec4(uColor, 1.0);
 }
`
```

Update from JavaScript:

```
function animate() {
 requestAnimationFrame(animate)
 material.uniforms.uTime.value = clock.getElapsedTime()
 renderer.render(scene, camera)
}
```

VARYINGS

Pass data from vertex to fragment shader:

```
let vertexShader = `
 varying vec2 vUv;

 void main() {
```

```
 vUv = uv;
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
`

let fragmentShader = `
 varying vec2 vUv;

 void main() {
 gl_FragColor = vec4(vUv, 0.0, 1.0);
 }
`
```

uv is texture coordinate (provided by Three.js).

vUv passes it to fragment shader.

Colors based on UV position.

ATTRIBUTES

Per-vertex data:

Built-in attributes:
- position - Vertex position (vec3)
- normal - Vertex normal (vec3)
- uv - Texture coordinates (vec2)

Custom attributes:

```
let colors = new Float32Array(vertexCount * 3)
geometry.setAttribute("color", new THREE.BufferAttribute(colors,
3))

let vertexShader = `
```

```
 attribute vec3 color;
 varying vec3 vColor;

 void main() {
 vColor = color;
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
`
```

GLSL DATA TYPES

float - Single number.

vec2 - Two numbers (x, y).

vec3 - Three numbers (x, y, z or r, g, b).

vec4 - Four numbers (x, y, z, w or r, g, b, a).

mat3 - 3x3 matrix.

mat4 - 4x4 matrix.

sampler2D - Texture.

GLSL OPERATIONS

Swizzling:

```
vec3 color = vec3(1.0, 0.5, 0.0);
vec2 rg = color.rg;
float r = color.r;
vec3 bgr = color.bgr;
```

Math functions:

```
float a = sin(x);
float b = cos(x);
float c = pow(x, 2.0);
float d = sqrt(x);
float e = abs(x);
float f = min(a, b);
float g = max(a, b);
float h = clamp(x, 0.0, 1.0);
float i = mix(a, b, 0.5);
float j = smoothstep(0.0, 1.0, x);
float k = fract(x);
```

ANIMATED SHADER

Color changing over time:

```
let material = new THREE.ShaderMaterial({
 uniforms: {
 uTime: { value: 0.0 }
 },
 vertexShader: `
 void main() {
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
 `,
 fragmentShader: `
 uniform float uTime;

 void main() {
 float r = sin(uTime) * 0.5 + 0.5;
 float g = sin(uTime + 2.0) * 0.5 + 0.5;
 float b = sin(uTime + 4.0) * 0.5 + 0.5;
```

```
 gl_FragColor = vec4(r, g, b, 1.0);
 }
 `
})
```

Update time:

```
function animate() {
 requestAnimationFrame(animate)
 material.uniforms.uTime.value = clock.getElapsedTime()
 renderer.render(scene, camera)
}
```

VERTEX DISPLACEMENT

Move vertices in shader:

```
let vertexShader = `
 uniform float uTime;

 void main() {
 vec3 pos = position;
 pos.z += sin(pos.x * 5.0 + uTime) * 0.2;
 pos.z += sin(pos.y * 5.0 + uTime) * 0.2;

 gl_Position = projectionMatrix * modelViewMatrix * vec4(pos,
1.0);
 }
`
```

Creates wave effect.

TEXTURES IN SHADERS

Pass texture as uniform:

```
let material = new THREE.ShaderMaterial({
 uniforms: {
 uTexture: { value: texture }
 },
 vertexShader: `
 varying vec2 vUv;

 void main() {
 vUv = uv;
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
 `,
 fragmentShader: `
 uniform sampler2D uTexture;
 varying vec2 vUv;

 void main() {
 vec4 color = texture2D(uTexture, vUv);
 gl_FragColor = color;
 }
 `
})
```

GRADIENT SHADER

Vertical gradient:

```
let fragmentShader = `
 varying vec2 vUv;

 void main() {
 vec3 colorA = vec3(1.0, 0.0, 0.0);
 vec3 colorB = vec3(0.0, 0.0, 1.0);
```

```
 vec3 color = mix(colorA, colorB, vUv.y);
 gl_FragColor = vec4(color, 1.0);
 }
`
```

NOISE IN SHADERS

Simple noise function:

```
float random(vec2 st) {
 return fract(sin(dot(st.xy, vec2(12.9898, 78.233))) *
43758.5453123);
}
```

Usage:

```
void main() {
 float n = random(vUv * 10.0);
 gl_FragColor = vec4(vec3(n), 1.0);
}
```

FRESNEL EFFECT

Edge glow effect:

```
let vertexShader = `
 varying vec3 vNormal;
 varying vec3 vViewPosition;

 void main() {
 vNormal = normalize(normalMatrix * normal);
 vec4 mvPosition = modelViewMatrix * vec4(position, 1.0);
 vViewPosition = -mvPosition.xyz;
```

```
 gl_Position = projectionMatrix * mvPosition;
 }
`

let fragmentShader = `
 varying vec3 vNormal;
 varying vec3 vViewPosition;

 void main() {
 vec3 viewDir = normalize(vViewPosition);
 float fresnel = pow(1.0 - dot(viewDir, vNormal), 2.0);
 gl_FragColor = vec4(vec3(fresnel), 1.0);
 }
`
```

RAWSHADERMATERIAL

No built-in uniforms or attributes:

```
let material = new THREE.RawShaderMaterial({
 uniforms: {
 projectionMatrix: { value: camera.projectionMatrix },
 modelViewMatrix: { value: new THREE.Matrix4() }
 },
 vertexShader: `
 precision mediump float;

 uniform mat4 projectionMatrix;
 uniform mat4 modelViewMatrix;

 attribute vec3 position;

 void main() {
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
 `,
```

```
fragmentShader: `
precision mediump float;

void main() {
gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
`
})
```

More control but more work.

SHADERMATERIAL OPTIONS

side - Which side to render.

material.side = THREE.DoubleSide

transparent:

material.transparent = true

wireframe:

material.wireframe = true

depthWrite and depthTest:

material.depthWrite = false
material.depthTest = false

SHADER CHUNKS

Three.js shader code can be reused:

```
let vertexShader = `
 #include <common>

 void main() {
 gl_Position = projectionMatrix * modelViewMatrix *
vec4(position, 1.0);
 }
`
```

common chunk includes utility functions.

EXTENDING BUILT-IN MATERIALS

Modify existing materials:

```
material.onBeforeCompile = function(shader) {
 shader.uniforms.uTime = { value: 0 }

 shader.vertexShader = shader.vertexShader.replace(
 "#include <begin_vertex>",
 `
 #include <begin_vertex>
 transformed.z += sin(transformed.x * 5.0 + uTime) * 0.2;
 `
 )
}
```

DEBUGGING SHADERS

Output values as colors:

```
void main() {
 gl_FragColor = vec4(vUv, 0.0, 1.0);
}
```

Visualize any value by outputting as color.

Check for compilation errors:

```
console.log(renderer.info.programs)
```

PRACTICE

Create a shader with animated color.

Pass uniforms from JavaScript.

Create a wave vertex shader.

Use textures in shaders.

Make a fresnel glow effect.

WHAT YOU LEARNED

You learned shader basics.

You learned uniforms and varyings.

You learned GLSL data types and functions.

You learned common shader effects.

You learned to debug shaders.

The next chapter teaches performance.
BOOK 18: THREE.JS COMPLETE
Chapter 20: Performance

WHY PERFORMANCE MATTERS

60 frames per second is the target.

Each frame has 16.67 milliseconds.

Slow performance causes stuttering.

Mobile devices have less power.

Good performance means wider audience.

MEASURING PERFORMANCE

Use stats.js:

npm install stats.js

```
import Stats from "stats.js"

let stats = new Stats()
document.body.appendChild(stats.dom)

function animate() {
 stats.begin()

 requestAnimationFrame(animate)
 renderer.render(scene, camera)
```

```
 stats.end()
}
```

Shows FPS, frame time, and memory.

RENDERER INFO

Check what renderer is doing:

```
function logInfo() {
 let info = renderer.info
 console.log("Draw calls:", info.render.calls)
 console.log("Triangles:", info.render.triangles)
 console.log("Geometries:", info.memory.geometries)
 console.log("Textures:", info.memory.textures)
}
```

Fewer draw calls and triangles is better.

DRAW CALLS

Each mesh is one draw call.

Each material change is one draw call.

Minimize draw calls for better performance.

Bad:

```
for (let i = 0; i < 1000; i++) {
 let mesh = new THREE.Mesh(geometry, material)
 scene.add(mesh)
```

```
}
```

1000 draw calls.

Good:

```
let instancedMesh = new THREE.InstancedMesh(geometry,
material, 1000)
scene.add(instancedMesh)
```

1 draw call.

INSTANCED MESH

Many objects with same geometry and material:

```
let geometry = new THREE.BoxGeometry(1, 1, 1)
let material = new THREE.MeshStandardMaterial({ color: 0xff0000
})
let count = 10000

let mesh = new THREE.InstancedMesh(geometry, material, count)

let matrix = new THREE.Matrix4()

for (let i = 0; i < count; i++) {
 matrix.setPosition(
 Math.random() * 100 - 50,
 Math.random() * 100 - 50,
 Math.random() * 100 - 50
 )
 mesh.setMatrixAt(i, matrix)
}

scene.add(mesh)
```

10000 objects in 1 draw call.

MERGE GEOMETRIES

Combine static objects:

```
import { mergeBufferGeometries } from "three/examples/jsm/utils/
BufferGeometryUtils.js"

let geometries = []

for (let i = 0; i < 100; i++) {
 let geo = new THREE.BoxGeometry(1, 1, 1)
 geo.translate(i * 2, 0, 0)
 geometries.push(geo)
}

let mergedGeometry = mergeBufferGeometries(geometries)
let mesh = new THREE.Mesh(mergedGeometry, material)

scene.add(mesh)
```

100 boxes in 1 draw call.

Cannot move individual boxes after merging.

GEOMETRY OPTIMIZATION

Reduce polygon count:

Use simpler shapes.

Decimate models in Blender.

Use BufferGeometry:

All built-in geometries are BufferGeometry.

More efficient than old Geometry class.

Index geometry:

Shared vertices reduce memory.

Most geometries are already indexed.

Dispose unused geometry:

geometry.dispose()

MATERIAL OPTIMIZATION

Reuse materials:

```
let material = new THREE.MeshStandardMaterial({ color: 0xff0000
})

let mesh1 = new THREE.Mesh(geometry1, material)
let mesh2 = new THREE.Mesh(geometry2, material)
```

Same material means same shader.

Use simpler materials:

MeshBasicMaterial is fastest.

MeshLambertMaterial is faster than Standard.

MeshStandardMaterial looks best but costs more.

Reduce material features:

material.flatShading = true

Fewer calculations per pixel.

TEXTURE OPTIMIZATION

Power of two sizes:

256, 512, 1024, 2048 pixels.

GPU handles these better.

Compress textures:

Use KTX2 with Basis Universal.

Smaller files and GPU memory.

Reduce texture size:

4096 is rarely needed.

1024 or 2048 is usually enough.

Mipmap filtering:

texture.generateMipmaps = true

Better quality at distance.

Enabled by default.

Dispose textures:

texture.dispose()

LIGHTING OPTIMIZATION

Fewer lights:

Each light adds calculations.

Combine into fewer lights.

Bake lighting:

Pre-calculate lighting in 3D software.

Apply as texture.

No runtime light calculations.

Disable shadows:

Shadows are expensive.

Use fake shadows (textured planes).

Only enable for important lights.

Reduce shadow map size:

```
light.shadow.mapSize.width = 512
light.shadow.mapSize.height = 512
```

FRUSTUM CULLING

Objects outside camera view are not rendered.

Enabled by default:

```
mesh.frustumCulled = true
```

Three.js skips rendering objects outside view.

LEVEL OF DETAIL

Use simpler models at distance:

```
let lod = new THREE.LOD()

let highDetail = new THREE.Mesh(highGeo, material)
let mediumDetail = new THREE.Mesh(mediumGeo, material)
let lowDetail = new THREE.Mesh(lowGeo, material)

lod.addLevel(highDetail, 0)
lod.addLevel(mediumDetail, 50)
lod.addLevel(lowDetail, 100)

scene.add(lod)
```

Update each frame:

lod.update(camera)

OBJECT POOLING

Reuse objects instead of creating new ones:

```
class ObjectPool {
 constructor(createFn, initialSize) {
 this.createFn = createFn
 this.available = []

 for (let i = 0; i < initialSize; i++) {
 this.available.push(createFn())
 }
 }

 get() {
 if (this.available.length > 0) {
 return this.available.pop()
 }
 return this.createFn()
 }

 release(obj) {
 this.available.push(obj)
 }
}
```

Usage:

```
let bulletPool = new ObjectPool(function() {
 return new THREE.Mesh(bulletGeo, bulletMaterial)
}, 100)
```

```
function shoot() {
 let bullet = bulletPool.get()
 bullet.position.copy(gun.position)
 scene.add(bullet)
}

function destroyBullet(bullet) {
 scene.remove(bullet)
 bulletPool.release(bullet)
}
```

OFFSCREEN RENDERING

Reduce resolution when not focused:

```
document.addEventListener("visibilitychange", function() {
 if (document.hidden) {
 renderer.setPixelRatio(0.5)
 } else {
 renderer.setPixelRatio(window.devicePixelRatio)
 }
})
```

PIXEL RATIO

Limit pixel ratio on high DPI screens:

```
renderer.setPixelRatio(Math.min(window.devicePixelRatio, 2))
```

4K screens do not need 4x pixels.

RENDER ON DEMAND

Only render when needed:

```
let needsRender = true

function requestRender() {
 needsRender = true
}

function animate() {
 requestAnimationFrame(animate)

 if (needsRender) {
 renderer.render(scene, camera)
 needsRender = false
 }
}

controls.addEventListener("change", requestRender)
```

WORKER THREADS

Heavy calculations off main thread:

```
let worker = new Worker("physics-worker.js")

worker.postMessage({ type: "step" })

worker.onmessage = function(event) {
 updateMeshPositions(event.data.positions)
}
```

Keep rendering smooth while calculating.

MEMORY MANAGEMENT

Dispose when removing objects:

```
function removeObject(obj) {
 scene.remove(obj)

 if (obj.geometry) {
 obj.geometry.dispose()
 }

 if (obj.material) {
 if (Array.isArray(obj.material)) {
 obj.material.forEach(m = > m.dispose())
 } else {
 obj.material.dispose()
 }
 }
}
```

Watch for memory leaks:

```
console.log(renderer.info.memory)
```

Numbers should not grow over time.

PROFILING

Use browser dev tools:

Performance tab shows frame times.

Memory tab shows allocations.

Chrome specific:

Type chrome://gpu in address bar.

Shows GPU information and issues.

MOBILE OPTIMIZATION

Reduce everything:

Fewer objects.

Smaller textures.

Simpler materials.

No shadows.

Detect mobile:

```
let isMobile = /iPhone|iPad|iPod|Android/
i.test(navigator.userAgent)

if (isMobile) {
 renderer.setPixelRatio(1)
 material = new THREE.MeshLambertMaterial({ color: 0xff0000 })
}
```

Touch events:

```
renderer.domElement.addEventListener("touchstart", onTouch)
```

PERFORMANCE CHECKLIST

Check draw calls (keep under 100).

Check triangle count.

Use instancing for repeated objects.

Merge static geometry.

Reuse materials.

Compress textures.

Limit shadow maps.

Use LOD for complex models.

Pool frequently created objects.

Dispose unused resources.

Profile regularly.

PRACTICE

Add stats.js to monitor FPS.

Convert multiple meshes to instanced mesh.

Implement object pooling.

Reduce texture sizes.

Profile and optimize a scene.

WHAT YOU LEARNED

You learned to measure performance.

You learned draw call reduction.

You learned geometry and material optimization.

You learned texture optimization.

You learned memory management.

The next chapter teaches Three.js programs.

BOOK 18: THREE.JS COMPLETE

Chapter 21: Three.js Programs

This chapter contains complete Three.js programs.

Each program demonstrates concepts from previous chapters.

Study these programs to understand how pieces fit together.

PROGRAM 1: ROTATING CUBE GALLERY

A gallery of colored cubes that rotate.

Demonstrates meshes, materials, animation loop.

index.html:

```
<!DOCTYPE html>
<html>
<head>
 <title>Cube Gallery</title>
 <style>
 body { margin: 0; overflow: hidden; }
 canvas { display: block; }
 </style>
</head>
<body>
 <script type="module" src="main.js"></script>
</body>
</html>
```

main.js:

```
import * as THREE from "three"
import { OrbitControls } from "three/examples/jsm/controls/
OrbitControls.js"

let scene = new THREE.Scene()
scene.background = new THREE.Color(0x111111)

let camera = new THREE.PerspectiveCamera(
 75,
 window.innerWidth / window.innerHeight,
 0.1,
 1000
)
camera.position.set(0, 5, 15)

let renderer = new THREE.WebGLRenderer({ antialias: true })
renderer.setSize(window.innerWidth, window.innerHeight)
renderer.setPixelRatio(Math.min(window.devicePixelRatio, 2))
document.body.appendChild(renderer.domElement)

let controls = new OrbitControls(camera, renderer.domElement)
```

```
controls.enableDamping = true

let cubes = []
let colors = [0xff0000, 0x00ff00, 0x0000ff, 0xffff00, 0xff00ff,
0x00ffff]

for (let i = 0; i < 6; i++) {
 let geometry = new THREE.BoxGeometry(2, 2, 2)
 let material = new THREE.MeshStandardMaterial({
 color: colors[i],
 metalness: 0.3,
 roughness: 0.4
 })
 let cube = new THREE.Mesh(geometry, material)

 let angle = (i / 6) * Math.PI * 2
 cube.position.x = Math.cos(angle) * 6
 cube.position.z = Math.sin(angle) * 6
 cube.position.y = 1

 scene.add(cube)
 cubes.push(cube)
}

let floorGeometry = new THREE.PlaneGeometry(30, 30)
let floorMaterial = new THREE.MeshStandardMaterial({
 color: 0x333333,
 roughness: 0.8
})
let floor = new THREE.Mesh(floorGeometry, floorMaterial)
floor.rotation.x = -Math.PI / 2
scene.add(floor)

let ambientLight = new THREE.AmbientLight(0xffffff, 0.4)
scene.add(ambientLight)

let directionalLight = new THREE.DirectionalLight(0xffffff, 0.8)
directionalLight.position.set(10, 20, 10)
scene.add(directionalLight)
```

```
let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)

 let elapsed = clock.getElapsedTime()

 for (let i = 0; i < cubes.length; i++) {
 cubes[i].rotation.x = elapsed * 0.5
 cubes[i].rotation.y = elapsed * 0.3
 cubes[i].position.y = 1 + Math.sin(elapsed * 2 + i) * 0.5
 }

 controls.update()
 renderer.render(scene, camera)
}

window.addEventListener("resize", function() {
 camera.aspect = window.innerWidth / window.innerHeight
 camera.updateProjectionMatrix()
 renderer.setSize(window.innerWidth, window.innerHeight)
})

animate()
```

PROGRAM 2: PARTICLE STARFIELD

A space starfield with twinkling stars.

Demonstrates particles, custom attributes, animation.

main.js:

```
import * as THREE from "three"

let scene = new THREE.Scene()
```

```
let camera = new THREE.PerspectiveCamera(
 75,
 window.innerWidth / window.innerHeight,
 0.1,
 1000
)
camera.position.z = 5

let renderer = new THREE.WebGLRenderer()
renderer.setSize(window.innerWidth, window.innerHeight)
document.body.appendChild(renderer.domElement)

let starCount = 5000
let positions = new Float32Array(starCount * 3)
let colors = new Float32Array(starCount * 3)
let sizes = new Float32Array(starCount)

for (let i = 0; i < starCount; i++) {
 let i3 = i * 3

 let radius = 50 + Math.random() * 450
 let theta = Math.random() * Math.PI * 2
 let phi = Math.acos(Math.random() * 2 - 1)

 positions[i3] = radius * Math.sin(phi) * Math.cos(theta)
 positions[i3 + 1] = radius * Math.sin(phi) * Math.sin(theta)
 positions[i3 + 2] = radius * Math.cos(phi)

 let colorValue = 0.8 + Math.random() * 0.2
 colors[i3] = colorValue
 colors[i3 + 1] = colorValue
 colors[i3 + 2] = colorValue + Math.random() * 0.1

 sizes[i] = Math.random() * 2
}

let geometry = new THREE.BufferGeometry()
geometry.setAttribute("position", new
THREE.BufferAttribute(positions, 3))
```

```
geometry.setAttribute("color", new THREE.BufferAttribute(colors,
3))
geometry.setAttribute("size", new THREE.BufferAttribute(sizes, 1))

let material = new THREE.PointsMaterial({
 size: 0.5,
 vertexColors: true,
 transparent: true,
 opacity: 0.8,
 sizeAttenuation: true
})

let stars = new THREE.Points(geometry, material)
scene.add(stars)

let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)

 let elapsed = clock.getElapsedTime()

 stars.rotation.y = elapsed * 0.02
 stars.rotation.x = elapsed * 0.01

 let sizeArray = geometry.attributes.size.array
 for (let i = 0; i < starCount; i++) {
 sizeArray[i] = (Math.sin(elapsed * 2 + i * 0.1) + 1) * 0.5 + 0.5
 }
 geometry.attributes.size.needsUpdate = true

 renderer.render(scene, camera)
}

window.addEventListener("resize", function() {
 camera.aspect = window.innerWidth / window.innerHeight
 camera.updateProjectionMatrix()
 renderer.setSize(window.innerWidth, window.innerHeight)
})
```

animate()

PROGRAM 3: INTERACTIVE SCENE

Click objects to change their color.

Demonstrates raycasting, event handling, materials.

main.js:

```
import * as THREE from "three"
import { OrbitControls } from "three/examples/jsm/controls/
OrbitControls.js"

let scene = new THREE.Scene()
scene.background = new THREE.Color(0x222222)

let camera = new THREE.PerspectiveCamera(
 75,
 window.innerWidth / window.innerHeight,
 0.1,
 1000
)
camera.position.set(0, 5, 10)

let renderer = new THREE.WebGLRenderer({ antialias: true })
renderer.setSize(window.innerWidth, window.innerHeight)
document.body.appendChild(renderer.domElement)

let controls = new OrbitControls(camera, renderer.domElement)
controls.enableDamping = true

let clickableObjects = []

let shapes = [
 new THREE.BoxGeometry(1, 1, 1),
```

```
 new THREE.SphereGeometry(0.6, 32, 32),
 new THREE.ConeGeometry(0.5, 1, 32),
 new THREE.TorusGeometry(0.5, 0.2, 16, 100),
 new THREE.CylinderGeometry(0.5, 0.5, 1, 32)
]

for (let x = -2; x <= 2; x++) {
 for (let z = -2; z <= 2; z++) {
 let geometry = shapes[Math.floor(Math.random() *
shapes.length)]
 let material = new THREE.MeshStandardMaterial({
 color: Math.random() * 0xffffff,
 metalness: 0.3,
 roughness: 0.4
 })
 let mesh = new THREE.Mesh(geometry, material)
 mesh.position.set(x * 2, 0.5, z * 2)
 scene.add(mesh)
 clickableObjects.push(mesh)
 }
}

let floorGeometry = new THREE.PlaneGeometry(20, 20)
let floorMaterial = new THREE.MeshStandardMaterial({ color:
0x444444 })
let floor = new THREE.Mesh(floorGeometry, floorMaterial)
floor.rotation.x = -Math.PI / 2
scene.add(floor)

let ambientLight = new THREE.AmbientLight(0xffffff, 0.5)
scene.add(ambientLight)

let directionalLight = new THREE.DirectionalLight(0xffffff, 0.8)
directionalLight.position.set(5, 10, 5)
scene.add(directionalLight)

let raycaster = new THREE.Raycaster()
let mouse = new THREE.Vector2()
let hoveredObject = null
```

```
window.addEventListener("mousemove", function(event) {
 mouse.x = (event.clientX / window.innerWidth) * 2 - 1
 mouse.y = -(event.clientY / window.innerHeight) * 2 + 1
})

window.addEventListener("click", function() {
 raycaster.setFromCamera(mouse, camera)
 let intersects = raycaster.intersectObjects(clickableObjects)

 if (intersects.length > 0) {
 let obj = intersects[0].object
 obj.material.color.setHex(Math.random() * 0xffffff)
 }
})

function animate() {
 requestAnimationFrame(animate)

 raycaster.setFromCamera(mouse, camera)
 let intersects = raycaster.intersectObjects(clickableObjects)

 if (hoveredObject) {
 hoveredObject.scale.set(1, 1, 1)
 hoveredObject = null
 }

 if (intersects.length > 0) {
 hoveredObject = intersects[0].object
 hoveredObject.scale.set(1.2, 1.2, 1.2)
 document.body.style.cursor = "pointer"
 } else {
 document.body.style.cursor = "default"
 }

 controls.update()
 renderer.render(scene, camera)
}

window.addEventListener("resize", function() {
 camera.aspect = window.innerWidth / window.innerHeight
```

```
 camera.updateProjectionMatrix()
 renderer.setSize(window.innerWidth, window.innerHeight)
})

animate()
```

PROGRAM 4: PHYSICS PLAYGROUND

Falling boxes with physics.

Demonstrates cannon-es physics, synchronization.

main.js:

```
import * as THREE from "three"
import * as CANNON from "cannon-es"
import { OrbitControls } from "three/examples/jsm/controls/
OrbitControls.js"

let scene = new THREE.Scene()
scene.background = new THREE.Color(0x87ceeb)

let camera = new THREE.PerspectiveCamera(
 75,
 window.innerWidth / window.innerHeight,
 0.1,
 1000
)
camera.position.set(0, 10, 20)

let renderer = new THREE.WebGLRenderer({ antialias: true })
renderer.setSize(window.innerWidth, window.innerHeight)
renderer.shadowMap.enabled = true
document.body.appendChild(renderer.domElement)

let controls = new OrbitControls(camera, renderer.domElement)
```

```
controls.enableDamping = true

let world = new CANNON.World()
world.gravity.set(0, -9.82, 0)

let groundMat = new CANNON.Material("ground")
let boxMat = new CANNON.Material("box")

let contactMat = new CANNON.ContactMaterial(groundMat,
boxMat, {
 friction: 0.4,
 restitution: 0.3
})
world.addContactMaterial(contactMat)

let groundShape = new CANNON.Plane()
let groundBody = new CANNON.Body({ mass: 0, material:
groundMat })
groundBody.addShape(groundShape)
groundBody.quaternion.setFromAxisAngle(new CANNON.Vec3(1, 0,
0), -Math.PI / 2)
world.addBody(groundBody)

let groundGeometry = new THREE.PlaneGeometry(50, 50)
let groundMaterial = new THREE.MeshStandardMaterial({ color:
0x228b22 })
let groundMesh = new THREE.Mesh(groundGeometry,
groundMaterial)
groundMesh.rotation.x = -Math.PI / 2
groundMesh.receiveShadow = true
scene.add(groundMesh)

let physicsObjects = []

function createBox(x, y, z, color) {
 let size = 0.5 + Math.random() * 0.5

 let geometry = new THREE.BoxGeometry(size * 2, size * 2, size *
2)
 let material = new THREE.MeshStandardMaterial({ color: color })
```

```
 let mesh = new THREE.Mesh(geometry, material)
 mesh.castShadow = true
 scene.add(mesh)

 let shape = new CANNON.Box(new CANNON.Vec3(size, size,
size))
 let body = new CANNON.Body({ mass: 1, material: boxMat })
 body.addShape(shape)
 body.position.set(x, y, z)
 body.angularVelocity.set(
 Math.random() * 2 - 1,
 Math.random() * 2 - 1,
 Math.random() * 2 - 1
 )
 world.addBody(body)

 physicsObjects.push({ mesh: mesh, body: body })
}

for (let i = 0; i < 50; i++) {
 createBox(
 (Math.random() - 0.5) * 10,
 5 + Math.random() * 20,
 (Math.random() - 0.5) * 10,
 Math.random() * 0xffffff
 )
}

let ambientLight = new THREE.AmbientLight(0xffffff, 0.5)
scene.add(ambientLight)

let directionalLight = new THREE.DirectionalLight(0xffffff, 0.8)
directionalLight.position.set(10, 20, 10)
directionalLight.castShadow = true
directionalLight.shadow.mapSize.width = 2048
directionalLight.shadow.mapSize.height = 2048
directionalLight.shadow.camera.near = 0.5
directionalLight.shadow.camera.far = 50
directionalLight.shadow.camera.left = -20
directionalLight.shadow.camera.right = 20
```

```
directionalLight.shadow.camera.top = 20
directionalLight.shadow.camera.bottom = -20
scene.add(directionalLight)

window.addEventListener("click", function() {
 createBox(
 (Math.random() - 0.5) * 5,
 15,
 (Math.random() - 0.5) * 5,
 Math.random() * 0xffffff
 )
})

let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)

 let delta = clock.getDelta()
 world.step(1 / 60, delta, 3)

 for (let obj of physicsObjects) {
 obj.mesh.position.copy(obj.body.position)
 obj.mesh.quaternion.copy(obj.body.quaternion)
 }

 controls.update()
 renderer.render(scene, camera)
}

window.addEventListener("resize", function() {
 camera.aspect = window.innerWidth / window.innerHeight
 camera.updateProjectionMatrix()
 renderer.setSize(window.innerWidth, window.innerHeight)
})

animate()
```

PROGRAM 5: SHADER WAVE

Animated wave using custom shader.

Demonstrates vertex shaders, uniforms, animation.

main.js:

```
import * as THREE from "three"
import { OrbitControls } from "three/examples/jsm/controls/
OrbitControls.js"

let scene = new THREE.Scene()
scene.background = new THREE.Color(0x000022)

let camera = new THREE.PerspectiveCamera(
 75,
 window.innerWidth / window.innerHeight,
 0.1,
 1000
)
camera.position.set(0, 5, 10)

let renderer = new THREE.WebGLRenderer({ antialias: true })
renderer.setSize(window.innerWidth, window.innerHeight)
document.body.appendChild(renderer.domElement)

let controls = new OrbitControls(camera, renderer.domElement)
controls.enableDamping = true

let geometry = new THREE.PlaneGeometry(20, 20, 128, 128)

let material = new THREE.ShaderMaterial({
 uniforms: {
 uTime: { value: 0 },
 uWaveHeight: { value: 0.5 },
 uWaveFrequency: { value: 2.0 },
 uColorA: { value: new THREE.Color(0x0066ff) },
```

```
uColorB: { value: new THREE.Color(0x00ffff) }
},
vertexShader: `
uniform float uTime;
uniform float uWaveHeight;
uniform float uWaveFrequency;

varying vec2 vUv;
varying float vElevation;

void main() {
vUv = uv;

vec3 pos = position;

float elevation = sin(pos.x * uWaveFrequency + uTime) *
uWaveHeight;
elevation += sin(pos.y * uWaveFrequency * 0.5 + uTime * 0.5) *
uWaveHeight * 0.5;

pos.z = elevation;
vElevation = elevation;

gl_Position = projectionMatrix * modelViewMatrix * vec4(pos,
1.0);
}
`,
fragmentShader: `
uniform vec3 uColorA;
uniform vec3 uColorB;

varying vec2 vUv;
varying float vElevation;

void main() {
float mixValue = (vElevation + 0.75) / 1.5;
vec3 color = mix(uColorA, uColorB, mixValue);

gl_FragColor = vec4(color, 1.0);
}
```

```
`,
 side: THREE.DoubleSide
})

let plane = new THREE.Mesh(geometry, material)
plane.rotation.x = -Math.PI / 2
scene.add(plane)

let clock = new THREE.Clock()

function animate() {
 requestAnimationFrame(animate)

 material.uniforms.uTime.value = clock.getElapsedTime()

 controls.update()
 renderer.render(scene, camera)
}

window.addEventListener("resize", function() {
 camera.aspect = window.innerWidth / window.innerHeight
 camera.updateProjectionMatrix()
 renderer.setSize(window.innerWidth, window.innerHeight)
})

animate()
```

WHAT YOU LEARNED

You learned complete Three.js program structure.

You learned to combine multiple concepts.

You learned practical implementation patterns.

You have reference code for your projects.

BOOK COMPLETE

You have completed Book 18: Three.js Complete.

You learned the foundations of 3D web graphics.

You can create scenes, add objects, apply materials.

You can animate, interact, and optimize.

You can create custom effects with shaders.

You can add physics and audio.

Keep building and experimenting.

The best way to learn is to create projects.